User and Reference Manual: All Parts

Release 4.1

23 October 2012

Built from 4.1-I-151

Contents

XVII Support Library 4137

Part I

General Introduction

Chapter 1

Introduction

CGAL Editorial Board

The goal of the CGAL Open Source Project is to provide *easy access to efficient and reliable geometric algorithms* in the form of a C++ library.

The Computational Geometry Algorithms Library offers data structures and algorithms like triangulations, Voronoi diagrams, Boolean operations on polygons and on polyhedra, arrangements of curves, mesh generation, geometry processing, convex hull algorithms, to name just a few.

All these data structures and algorithms operate on geometric objects like points and segments, and perform geometric tests on them. These objects and predicates are regrouped in CGAL *Kernels*.

Finally, the CGAL *Support Library* offers geometric object generators and spatial sorting functions, as well as a matrix search framework and a solver for linear and quadratic programs. It further offers interfaces to third party software such as the GUI libraries Qt, Geomview, and the Boost Graph Library.

1.1 Organization of the Manual

This manual is organized in several parts covering the many domains of computational geometry. Each part consists of several chapters, and each chapter is split into a *user manual* and a *reference manual*. The user manual gives the general idea and comes with examples. The reference manual presents the API of the various classes and functions.

The manual has a table of contents, and an index, as well as a package overview, which gives a short paragraph what the package is about, what license it has, and on which other packages it depends. It further provides links to precompiled demo programs for the Windows platform.

1.2 Demos and Examples

In the distribution of the library you find the two directories *demo* and *examples*. They contain subdirectories for the CGAL packages. The demos use third party libraries for the graphical user interface. The examples don't have this dependency and most examples are refered to in the user manual.

1.3 Hello World

In this section we will take a closer look at three CGAL example programs, all of them computing the 2D convex hull of a set of points.

1.3.1 Points in a built-in array

In the first example we have an array of five points. As the convex hull of these points is a subset of the input it is safe to provide an array for storing the result which has the same size.

```
#include <iostream>
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/convex_hull_2.h>

typedef CGAL::Exact_predicates_inexact_constructions_kernel K;
typedef K::Point_2 Point_2;

int main()
{
  Point_2 points[5] = { Point_2(0,0), Point_2(10,0), Point_2(10,10),
   Point_2(6,5), Point_2(4,1) };
  Point_2 result[5];

  Point_2 *ptr = CGAL::convex_hull_2( points, points+5, result );
  std::cout <<  ptr - result << " points on the convex hull" <<
   std::endl;
  return 0;
}
```

File: `examples/Convex_hull_2/array_convex_hull_2.cpp`

All CGAL header files are in the subdirectory “include/CGAL”. All CGAL classes and functions are in the namespace “CGAL”. The geometric primitives, like the point type, are defined in a kernel. CGAL comes with several kernels, and as the convex hull algorithm only makes comparisons of coordinates and orientation tests of input points, we can choose a kernel that provides exact predicates, but no exact geometric construction.

The convex hull function takes three arguments, the start and past-the-end pointer for the input, and the start pointer of the array for the result. The function returns the pointer into the result array just behind the last convex hull point written, so the pointer difference tells us how many points are on the convex hull.

1.3.2 Points in a STL vector

In the second example we replace the built-in array by a *std::vector* of the Standard Template Library.

```
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/convex_hull_2.h>

#include <vector>
```

```
typedef CGAL::Exact_predicates_inexact_constructions_kernel K;
typedef K::Point_2 Point_2;
typedef std::vector<Point_2> Points;

int main()
{
  Points points, result;
  points.push_back(Point_2(0,0));
  points.push_back(Point_2(10,0));
  points.push_back(Point_2(10,10));
  points.push_back(Point_2(6,5));
  points.push_back(Point_2(4,1));


  CGAL::convex_hull_2( points.begin(), points.end(),
   std::back_inserter(result) );
  std::cout << result.size() << " points on the convex hull" <<
   std::endl;
  return 0;
}
```

File: examples/Convex_hull_2/vector_convex_hull_2.cpp

We put some points in the vector calling the *push_back()* method of the *std::vector* class.

We then call the convex hull function. The first two arguments, *points.begin()* and *points.end()* are *iterators*, which are a generalization of pointers: they can be dereferenced and incremented. The convex hull function is *generic* in the sense that it takes as input whatever can be dereferenced and incremented.

The third argument is where the result gets written to. In the previous example we provided a pointer to allocated memory. The generalization of such a pointer is the *output iterator*, which allows to increment and assign a value to the dereferenced iterator. In this example we start with an empty vector which grows as needed. Therefore, we cannot simply pass it *result.begin()*, but an output iterator generated by the helper function *std::back_inserter(result)*. This output iterator does nothing when incremented, and calls *result.push_back(..)* on the assignment.

1.3.3 Points in Streams

The last example program reads a sequence of points from standard input *std::cin* and writes the points on the convex hull to standard output *std::cout*.

Instead of storing the points in a container such as an *std::vector*, and passing the begin/end iterator of the vector to the convex hull function, we use helper classes that turn file pointers into iterators.

```
#include <iostream>
#include <iterator>
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/convex_hull_2.h>

typedef CGAL::Exact_predicates_inexact_constructions_kernel K;
typedef K::Point_2 Point_2;
```

```
int main()
{
  std::istream_iterator< Point_2 >  input_begin( std::cin );
  std::istream_iterator< Point_2 >  input_end;
  std::ostream_iterator< Point_2 >  output( std::cout, "\n" );
  CGAL::convex_hull_2( input_begin, input_end, output );
  return 0;
}
```

File: examples/Convex_hull_2/iostream_convex_hull_2.cpp

In the example code you see input and output stream iterators templated with the point type. A *std::istream_iterator<Point_2>* hence allows to traverse a sequence of objects of type *Point_2*, which come from standard input as we pass *std::cin* to the constructor of the iterator. The variable *input_end* denotes end-of-file.

A *std::ostream_iterator<Point_2>* is an output iterator, that is an iterator to which, when dereferenced, we can assign a value. When such an assignment to the output iterator happens somewhere inside the convex hull function, the iterator just writes the assigned point to standard output, because the iterator was constructed with *std::cout*.

The call to the convex hull function takes three arguments, the input iterator range, and the output iterator to which the result gets written.

If you know the STL, the Standard Template Library, the above makes perfect sense, as this is the way the STL decouples algorithms from containers. If you don't know the STL, you maybe better first familiarize yourself with its basic ideas.

1.4 Further Reading

We also recommend the standard text books by Josuttis [Jos99], or Austern [Aus98] for the STL and its notion of *concepts* and *models*.

Other resources for CGAL are the tutorials at http://www.cgal.org/Tutorials/ and the user support page at http://www.cgal.org/.

Introduction
Reference Manual

CGAL Editorial Board

This chapter contains no Reference Manual pages.

Chapter 2

Preliminaries

CGAL Editorial Board

This chapter lists the licenses under which the CGAL datastructures and algorithms are distributed. The chapter further explains how to control inlining, thread safety, code deprecation, checking of pre- and postconditions, and how to alter the failure behavior.

2.1 License Issues

CGAL is distributed under a dual license scheme, that is under the GPL/LGPL open source license, as well as under commercial licenses.

CGAL consists of different parts covered by different open source licenses. In this section we explain the essence of the different licenses, as well as the rationale why we have chosen them.

The fact that CGAL is Open Source software does not mean that users are free to do whatever they want with the software. Using the software means to accept the license, which has the status of a contract between the user and the owner of the CGAL software.

2.1.1 GPL

The GPL is an Open Source license that, if you distribute your software based on GPLed CGAL data structures,you are obliged to distribute the source code of your software under the GPL.

The exact license terms can be found at the Free Software Foundation web site: `http://www.gnu.org/copyleft/gpl.html`.

2.1.2 LGPL

The LGPL is an Open Source license that obliges you to distribute modifications you make on CGAL software accessible to the users. In contrast to the GPL , there is no obligation to make the source code of software you build on top of LGPLed CGAL data structures

The exact license terms can be found at the Free Software Foundation web site: `http://www.gnu.org/copyleft/lesser.html`.

2.1.3 Rationale of the License Choice

We have chosen the GPL and the LGPL as they are well known and well understood open source licenses. The former restricts commercial use, and the latter allows to promote software as de facto standard so that people can build new higher level data structures on top.

Therefore, the packages forming a foundation layer are distributed under the LGPL, and the higher level packages under the GPL. The package overview states for each package under which license it is distributed.

2.1.4 Commercial Licenses

Users who cannot comply to the Open Source license terms can buy individual data structures under various commercial licenses from GeometryFactory: `http://www.geometryfactory.com/`. License fees paid by commercial customers are reinvested in R&D performed by the CGAL project partners, as well as in evolutive maintenance.

2.2 Marking of Special Functionality

In this manual you will encounter sections marked as follows.

2.2.1 Advanced Features

Some functionality is considered more advanced, for example because it is relatively low-level, or requires special care to be properly used.

advanced

Such functionality is identified this way in the manual.

advanced

2.2.2 Debugging Support Features

Usually related to advanced features that for example may not guarantee class invariants, some functionality is provided that helps debugging, for example by performing invariants checks on demand.

debugging support

Such functionality is identified this way in the manual.

debugging support

2.2.3 Deprecated Code

Sometimes, the CGAL project decides that a feature is deprecated. This means that it still works in the current release, but it will be removed in the next, or a subsequent release. This can happen when we have found a better way to do something, and we would like to reduce the maintenance cost of CGAL at some point in the future. There is a trade-off between maintaining backward compatibility and implementing new features more easily.

In order to help users manage the changes to apply to their code, we attempt to make CGAL code emit warnings when deprecated code is used. This can be done using some compiler specific features. Those warnings can be disabled by defining the macro *CGAL_NO_DEPRECATION_WARNINGS*. On top of this, we also provide a macro, *CGAL_NO_DEPRECATED_CODE*, which, when defined, disables all deprecated features. This allows users to easily test if their code relies on deprecated features.

deprecated

Such functionality is identified this way in the manual.

deprecated

2.3 Namespace CGAL

All names introduced by CGAL, especially those documented in these manuals, are in a namespace called *CGAL*, which is in global scope. A user can either qualify names from CGAL by adding *CGAL::*, e.g., *CGAL::Point_2< CGAL::Exact_predicates_inexact_constructions_kernel >*, make a single name from CGAL visible in a scope via a *using* statement, e.g., *using CGAL::Point_2;*, and then use this name unqualified in this scope, or even make all names from namespace *CGAL* visible in a scope with *using namespace CGAL;*. The latter, however, is likely to give raise to name conflicts and is therefore not recommended.

2.4 Inclusion Order of Header Files

Not all compilers fully support standard header names. CGAL provides workarounds for these problems in *CGAL/basic.h*. Consequently, as a golden rule, you should always include *CGAL/basic.h* first in your programs (or *CGAL/Cartesian.h*, or *CGAL/Homogeneous.h*, since they include *CGAL/basic.h* first).

2.5 Thread Safety

CGAL is progressively being made thread-safe. The guidelines which are followed are:

- it should be possible to use different objects in different threads at the same time (of the same type or not),
- it is not safe to access the same object from different threads at the same time, unless otherwise specified in the class documentation.

If the macro *CGAL_HAS_THREADS* is not defined, then CGAL assumes it can use any thread-unsafe code (such as static variables). By default, this macro is not defined, unless *BOOST_HAS_THREADS* or *_OPENMP* is defined. It is possible to force its definition on the command line, and it is possible to prevent its default definition by setting *CGAL_HAS_NO_THREADS* from the command line.

2.6 C++11 Support

CGAL is based on the C++ standard released in 1998 (and later refined in 2003). A new major version of this standard has been released, and is refered to as C++11. Some compilers and standard library implementations already provide some of the functionality of this new standard. For example, g++ provides a command-line switch (*-std=c++0x* or *-std=c++11* depending on the compiler version) which enables some of those features.

CGAL attempts to support this mode progressively, and already makes use of some of these features if they are available, although no extensive support has been implemented yet.

2.7 Functor Return Types

CGAL functors support the result_of protocol. If a functor *F* has the same return type across all overloads of *operator()*, the nested type *F::result_type* is defined to be that type. Otherwise the return type of calling the functor with an argument of type *Arg* can be accessed through *boost::result_of<F(Arg)>::type*.

2.8 Checks

Much of the CGAL code contains checks. For example, all checks used in the kernel code are prefixed by *CGAL_KERNEL*. Other packages have their own prefixes, as documented in the corresponding chapters. Some are there to check if the kernel behaves correctly, others are there to check if the user calls kernel routines in an acceptable manner.

There are five types of checks. The first three are errors and lead to a halt of the program if they fail. The fourth only leads to a warning, and the last one is compile-time only.

Preconditions check if the caller of a routine has called it in a proper fashion. If such a check fails it is the responsibility of the caller (usually the user of the library).

Postconditions check if a routine does what it promises to do. If such a check fails it is the fault of this routine, so of the library.

Assertions are other checks that do not fit in the above two categories.

Warnings are checks for which it is not so severe if they fail.

Static assertions are compile-time assertions, used e.g. to verify the values of compile-time constants or compare types for (in)equality.

By default, all of these checks are performed. It is however possible to turn them off through the use of compile time switches. For example, for the checks in the kernel code, these switches are the following: *CGAL_KERNEL_NO_PRECONDITIONS*, *CGAL_KERNEL_NO_POSTCONDITIONS*, *CGAL_KERNEL_NO_ASSERTIONS* and *CGAL_KERNEL_NO_WARNINGS*. So, in order to compile the file `foo.cpp` with the postcondition checks off, you can do:

```
CC -DCGAL_KERNEL_NO_POSTCONDITIONS foo.cpp
```

This is also preferably done by modifying your makefile by adding *-DCGAL_KERNEL_NO_POSTCONDITIONS* to the *CXXFLAGS* variable.

The name *KERNEL* in the macro name can be replaced by a package specific name in order to control assertions done in a given package. This name is given in the documentation of the corresponding package, in case it exists.

Note that global macros can also be used to control the behavior over the whole CGAL library:

- *CGAL_NO_PRECONDITIONS*,
- *CGAL_NO_POSTCONDITIONS*,
- *CGAL_NO_ASSERTIONS*,
- *CGAL_NO_WARNINGS* and
- *CGAL_NDEBUG*.

Setting the macro *CGAL_NDEBUG* disables all checks. Note that the standard flag *NDEBUG* sets *CGAL_NDEBUG*, but it also affects the standard *assert* macro. This way, adding *-DCGAL_NDEBUG* to your compilation flags removes absolutely all checks. This is the default recommended setup for performing timing benchmarks for example.

Not all checks are on by default. The first four types of checks can be marked as expensive or exactness checks (or both). These checks need to be turned on explicitly by supplying one or both of the compile time switches *CGAL_KERNEL_CHECK_EXPENSIVE* and *CGAL_KERNEL_CHECK_EXACTNESS*.

Expensive checks are, as the word says, checks that take a considerable time to compute. Considerable is an imprecise phrase. Checks that add less than 10 percent to the execution time of the routine they are in are not expensive. Checks that can double the execution time are. Somewhere in between lies the border line. Checks that increase the asymptotic running time of an algorithm are always considered expensive. Exactness checks are checks that rely on exact arithmetic. For example, if the intersection of two lines is computed, the postcondition of this routine may state that the intersection point lies on both lines. However, if the computation is done with doubles as number type, this may not be the case, due to round off errors. So, exactness checks should only be turned on if the computation is done with some exact number type.

By definition, static assertions are both inexpensive and unaffected by precision management. Thus, the categories do not apply for static assertions.

2.8.1 Altering the Failure Behavior

As stated above, if a postcondition, precondition or assertion is violated, an exception is thrown, and if nothing is done to catch it, the program will abort. This behavior can be changed by means of the following function.

#include <CGAL/assertions_behaviour.h>

Failure_behaviour   *set_error_behaviour(Failure_behaviour eb)*

The parameter should have one of the following values.

enum Failure_behaviour { *ABORT, EXIT, EXIT_WITH_SUCCESS, CONTINUE, THROW_EXCEPTION*}*;*

The *THROW_EXCEPTION* value is the default, which throws an exception.

If the *EXIT* value is set, the program will stop and return a value indicating failure, but not dump the core. The *CONTINUE* value tells the checks to go on after diagnosing the error. Note that since CGAL 3.4, *CONTINUE* has the same effect as *THROW_EXCEPTION* for errors (but it keeps its meaning for warnings), it is not possible anymore to let assertion failures simply continue (except by totally disabling them).

advanced

If the *EXIT_WITH_SUCCESS* value is set, the program will stop and return a value corresponding to successful execution and not dump the core.

advanced

The value that is returned by *set_error_behaviour* is the value that was in use before.

For warnings there is a separate routine, which works in the same way. The only difference is that for warnings the default value is *CONTINUE*.

Failure_behaviour   *set_warning_behaviour(Failure_behaviour eb)*

2.8.2 Control at a Finer Granularity

The compile time flags as described up to now all operate on the whole library. Sometimes you may want to have a finer control. CGAL offers the possibility to turn checks on and off with a bit finer granularity, namely the module in which the routines are defined. The name of the module is to be appended directly after the CGAL prefix. So, the flag *CGAL_KERNEL_NO_ASSERTIONS* switches off assertions in the kernel only, the flag *CGAL_CH_CHECK_EXPENSIVE* turns on expensive checks in the convex hull module. The name of a particular module is documented with that module.

advanced

2.8.3 Customizing how Errors are Reported

Normally, error messages are written to the standard error output. It is possible to do something different with them. To that end you can register your own handler. This function should be declared as follows.

```
void        my_failure_function(
                const char *type,
                const char *expression,
                const char *file,
                int line,
                const char *explanation)
```

Your failure function will be called with the following parameters. *type* is a string that contains one of the words precondition, postcondition, assertion or warning. The parameter *expression* contains the expression that was violated. *file* and *line* contain the place where the check was made. The *explanation* parameter contains an explanation of what was checked. It can be *NULL*, in which case the *expression* is thought to be descriptive enough.

There are several things that you can do with your own handler. You can display a diagnostic message in a different way, for instance in a pop up window or to a log file (or a combination). You can also implement a different policy on what to do after an error. For instance, you can throw an exception or ask the user in a dialog whether to abort or to continue. If you do this, it is best to set the error behavior to *CONTINUE*, so that it does not interfere with your policy.

You can register two handlers, one for warnings and one for errors. Of course, you can use the same function for both if you want. When you set a handler, the previous handler is returned, so you can restore it if you want.

#include <CGAL/assertions.h>

Failure_function *set_error_handler(Failure_function handler)*

Failure_function *set_warning_handler(Failure_function handler)*

Example

```
#include <CGAL/assertions.h>

 void my_failure_handler(
     const char *type,
     const char *expr,
     const char* file,
     int line,
     const char* msg)
 {
     /* report the error in some way. */
 }

 void foo()
 {
     CGAL::Failure_function prev;
     prev = CGAL::set_error_handler(my_failure_handler);
     /* call some routines. */
     CGAL::set_error_handler(prev);
 }
```

advanced

2.9 Identifying the Version of CGAL

#include <CGAL/config.h>

Every release of CGAL defines the following preprocessor macros:

CGAL_VERSION – a textual description of the current release (e.g., or 3.3 or 3.2.1 or 3.2.1-I-15), and

CGAL_VERSION_STR – same as `CGAL_VERSION` but as a string constant token, and

CGAL_VERSION_NR – a numerical description of the current release such that more recent releases have higher number.

More precisely, it is defined as `1MMmmbiiii`, where `MM` is the major release number (e.g. 03), `mm` is the minor release number (e.g. 02), `b` is the bug-fix release number (e.g. 0), and `iiii` is the internal release number (e.g. 0001). For public releases, the latter is defined as 1000. Examples: for the public release 3.2.4 this number is 1030241000; for internal release 3.2-I-1, it is 1030200001. Note that this scheme was modified around 3.2-I-30.

CGAL_VERSION_NUMBER(M,m,b) – a function macro computing the version number macro from the M.m.b release version. Note that the internal release number is dropped here. Example: `CGAL_VERSION_NUMBER(3,2,4)` is equal to 1030241000.

advanced

2.10 Compile-time Flags to Control Inlining

Making functions inlined can, at times, improve the efficiency of your code. However this is not always the case and it can differ for a single function depending on the application in which it is used. Thus CGAL defines a set of compile-time macros that can be used to control whether certain functions are designated as inlined functions or not. The following table lists the macros and their default values, which are set in one of the CGAL include files.

macro name	default
CGAL_KERNEL_INLINE	inline
CGAL_KERNEL_MEDIUM_INLINE	
CGAL_KERNEL_LARGE_INLINE	
CGAL_MEDIUM_INLINE	inline
CGAL_LARGE_INLINE	
CGAL_HUGE_INLINE	

If you wish to change the value of one or more of these macros, you can simply give it a new value when compiling. For example, to make functions that use the macro *CGAL_KERNEL_MEDIUM_INLINE* inline functions, you should set the value of this macro to `inline` instead of the default blank.

Note that setting inline manually is very fragile, especially in a template context. It is usually better to let the compiler select by himself which functions should be inlined or not.

advanced

Preliminaries
Reference Manual

CGAL Editorial Board

This chapter contains no Reference Manual pages.

Chapter 3

Installation

Eric Berberich, Joachim Reichel, and Fernando Cacciola

Contents

3.1 Introduction

This document describes how to install CGAL on Windows, Unix-like systems, and MacOS X.

Ideally, setting up CGAL amounts to:

```
cd  CGAL-4.1                          # go to CGAL directory
cmake .                               # configure CGAL
make                                  # build the CGAL libraries
```

Compiling an example shipped with CGAL is similar simple:

```
cd examples/Straight_skeleton_2       # go to an example directory
cmake -DCGAL_DIR=$HOME/CGAL-4.1 .     # configure the examples
make                                  # build the examples
```

Compiling an own non-shipped program is also close:

```
cd /path/to/program
cgal_create_CMakeLists -s executable
cmake -DCGAL_DIR=$HOME/CGAL-4.1 .
make
```

where the second line creates a `CMakeLists.txt` file (check its options in Section 3.15.1 for various details).

In a less ideal world, you probably have to install CMake, a makefile generator, and third party libraries. That is what this manual is about.

3.2 Prerequisites

Installing CGAL requires a few components to be installed ahead: a supported compiler (see Section 3.5, CMake, BOOST, and at least GMP, and MPFR; see Section 3.8 for more details on essential third party software.

3.3 OS-shipped CGAL

Some operating systems with package managers offer CGAL and its essential third party software through the manager, for instance, Mac OS X, or some Linux distribution (e.g. Debian).

3.3.1 CGAL on MAC OS X

For instance, use macports in the following way:

```
sudo port install cgal
```

or if Qt4 demos are desired

```
sudo port install cgal +qt4 +universal +demos
```

The setup is similar for homebrew.

3.3.2 CGAL on Linux

For instance in debian/Ubuntu, use apt-get in the following way:

```
sudo apt-get install libcgal-dev
```

To get the demos use

```
sudo apt-get install libcgal-demo
```

Check the CGAL-FAQ for source repository of newest releases.

On other distributions, please consult your package manager documentation.

3.4 Downloading CGAL

You can obtain the CGAL library from `http://www.cgal.org/download.html` and install it yourself.

After you have downloaded the file `CGAL-4.1.tar.gz` containing the CGAL sources, you have to unpack it. Under a Unix-like shell, use the command:

```
tar xzf CGAL-4.1.tar.gz
```

When you are on Windows you may download and run `CGAL-4.1-Setup.exe`. It is a self extracting executable that installs the CGAL source, and that allows you to select and download some precompiled third party libraries.

In both cases the directory `CGAL-4.1` will be created. This directory contains the following subdirectories:

directory	**contents**
`auxiliary`	precompiled GMPand MPFR for Windows
`cmake/modules`	modules for finding and using libraries
`config`	configuration files for install script
`demo`	demo programs (most of them need QT, geomview or other third-party products)
`doc_html`	documentation (HTML)
`examples`	example programs
`include`	header files
`scripts`	some useful scripts (e.g. for creating CMakeLists.txt files)
`src`	source files

The directories `include/CGAL/CORE` and `src/CGALCore` contain a distribution of the CORE library[1] version 1.7 for dealing with algebraic numbers. CORE is not part of CGAL and has its own license.

The directory `include/CGAL/OpenNL` contains a distribution of the Open Numerical Library which provides solvers for sparse linear systems, especially designed for the Computer Graphics community. OPENNL is not part of CGAL and has its own license.

The only documentation shipped with CGAL sources is the present installation manual. The CGAL manual must be downloaded separately from `http://www.cgal.org/download.html`.

3.5 Supported Compilers

In order to build the CGAL libraries, you need a C++ compiler. CGAL 4.1 is supported for the following compilers/operating systems:

compiler	**operating system**
GNU `g++` 3.4 or later[1]	Linux / MacOS X MS Windows 95/98/2000/XP/NT4
INTEL `C++` 11.0 or later[2]	Linux
MS Visual `C++` 9.0, 10.0 (VISUAL STUDIO 2008 AND 2010)[3]	MS Windows 95/98/2000/XP/NT4/Vista/7

[1]`http://www.cs.nyu.edu/exact/`
[1]`http://gcc.gnu.org/`
[2]`http://software.intel.com/en-us/intel-compilers/`
[3]`http://msdn.microsoft.com/en-us/vstudio/`

3.6 Configuring CGAL with CMake

In order to configure, build, and install the CGAL libraries, examples and demos, you need CMake, a cross-platform "makefile generator". If CMake is not installed already you can obtain it from `http://www.cmake.org/`. CMake version 2.6.2 or higher is required. On Windows, CMake version 2.8.6 or higher is required, for a proper support of DLL's generation. This manual explains only those features of CMake which are needed in order to build CGAL. Please refer to the CMake documentation at `http://www.cmake.org/` for further details.

Before building CGAL you have to choose the compiler/linker, set compiler and linker flags, specify which third-party libraries you want to use and where they can be found, and which CGAL libraries you want to build. Gathering all this information is called *configuration*. The end of the process is marked by the generation of a makefile or a Visual C++ solution and project file that you can use to build CGAL.

3.6.1 Configuring CGAL with the CMake GUI

The simplest way to start the configuration is to run the graphical user interface of CMake. We recommend to use `cmake-gui`. It is available on many platforms as of CMake version 2.6. You must pass as argument the root directory of CGAL. For example:

```
cd  CGAL-4.1
cmake-gui . # Notice the dot to indicate the current directory.
```

After `cmake-gui` opens, press 'Configure'. A dialog will pop up and you will have to choose what shall gets generated. After you have made your choice and pressed 'Finish', you will see the output of configuration tests in the lower portion of the application. When these tests are done, you will see many red entries in the upper portion of the application. Just ignore them and press 'Configure'. By now CMake should have found many libraries and have initialized variables. If you still find red entries, you have to provide the necessary information. This typically happens if you have installed software at non-standard locations. Providing information and pressing 'Configure' goes on until all entries are grayed. You are now ready to press 'Generate'. Once this is done, you can quit `cmake-gui`.

3.6.2 Configuring CGAL with the cmake command-line tool

Alternatively, you can run the command-line tool called `cmake`. You pass as argument the root directory of CGAL. For example:

```
cd  CGAL-4.1
cmake . # Notice the dot to indicate the current directory.
```

The very first thing CMake does is to detect the compiler to use. This detection is performed by a special CMake module called a *generator*. A CMake generator understands the build requirements for a particular compiler/linker and generates the necessary files for that. For example, the *UNIX Makefiles* generator understands the GNU chain of tools (g++, ld etc.) and produces makefiles, which can be used to build a target by a simple call to `make`. Likewise, the *Visual Studio 2010* generator produces solution and project files and can be manually launched in the VS IDE to build the target.

Each platform has a default generator, so you only need to select one when the default is not what you want. For example, under Windows, it is possible to generate *NMakefiles* instead of Visual Studio project files in order to build the library with `nmake`. Running `cmake` with no parameters in a command-line prints the list of available generators supported by your platform and CMake version. If the generator you need is not listed there, you can try a newer CMake version, as generators are hardcoded into CMake, and additional generators are added with each release.

Since the choice of the generator determines the type of build files to generate, in some cases you choose a particular generator as a mean to choose a specific compiler (because they use different build files). For example, the following generates solution files for use in Visual C++ 11.0 on a 64bit machine:

```
cd CGAL-4.1
cmake -G"Visual Studio 11 Win64" .
```

In other cases, however, the generator doesn't directly identify a specific compiler but a chain of tools. For example, the `UNIX Makefiles` generator produces `makefiles` that call some auto-detected command-line compiler, like g++. If you need the makefiles to use a different compiler, you need to specify the desired compiler in the call to CMake, as in this example:

```
cd CGAL-4.1
cmake -DCMAKE_CXX_COMPILER:FILEPATH=g++-4.7 .
```

CMake maintains configuration parameters in so-called *cmake variables*, like the `CMAKE_CXX_COMPILER` in the example above. These variables *are not environment variables* but *CMake variables*. Some of the CMake variables represent user choices, such as `WITH_examples` or `CMAKE_BUILD_TYPE=Release`, while others indicate the details of a third-party library, such as `Boost_INCLUDE_DIR` or the compiler flags to use, such as `CMAKE_CXX_FLAGS`.

The command line tool `cmake` accepts CMake variables as arguments of the form `-D<VAR>:<TYPE>=<VALUE>`, as in the example above, but this is only useful if you already know which variables need to be explicitly defined.

advanced

CMake keeps the variables that a user can manipulate in a so-called *CMake cache*, a simple text file named `CMakeCache.txt`, whose entries are of the form VARIABLE:TYPE=VALUE. Advanced users can manually edit this file, instead of going through the interactive configuration session.

advanced

The configuration process not only determines the location of the required dependencies, it also dynamically generates a `compiler_config.h` file, which encodes the properties of your system and a special file named `CGALConfig.cmake`, which is used to build programs using CGAL. The purpose of this file is explained below.

3.7 CGAL Libraries

CGAL is split into five libraries. During configuration, you can select the libraries that you would like to build by setting a CMake variable of the form `WITH_<library>`. By default all are switched `ON`. All activated libraries are build after configuration; see 3.11

We next list the libraries and essential 3rd party software (see 3.8) for each library:

library	CMake variable	functionality	dependencies
`CGAL`	*none*	Main library	GMP, MPFR, BOOST (headers) and Boost.Thread (library)
`CGAL_Core`	`WITH_CGAL_Core`	The CORE library for algebraic numbers.[15]	GMP and MPFR
`CGAL_ImageIO`	`WITH_CGAL_ImageIO`	Utilities to read and write image files	OPENGL, ZLIB, VTK(optional)
`CGAL_Qt3`	`WITH_CGAL_Qt3`	*CGAL::Qt_widget* used by QT3-based demos	QT3 and OPENGL
`CGAL_Qt4`	`WITH_CGAL_Qt4`	*QGraphicsView* support for QT4-based demos	QT4 and OPENGL

3.7.1 Debug vs. Release

The CMake variable `CMAKE_BUILD_TYPE` indicates how to build the libraries. It accepts the values `Release` or `Debug`. The default is `Release` and should be kept, unless you want to debug your program.

This is not an issue for solution/project files, as there the user selects the build type from within the IDE.

3.7.2 Static vs. Shared Libraries

Shared libraries, also called *dynamic-link libraries*, are built by default (`.dll` on Windows, `.so` on Linux, `.dylib` on MacOS). You can choose to produce static libraries instead by setting the CMake variable `BUILD_SHARED_LIBS` to `FALSE`. If you use `cmake-gui`, a tick box for that variable is available to set it.

These setting affect the variants of third-party libraries (see next section) selected whenever the choice is available.

3.8 Essential Third Party Libraries

The focus of CGAL is on geometry, and we rely on other highly specialized libraries and software for non-geometric issues, for instance, for numeric solvers, or visualization. We first list software that is essential to build (all) libraries of CGAL, that is, this software must be found during the configuration of CGAL for an actived library of CGAL (i.e. `WITH_<library>=ON`); see 3.16.4 to specify the location of 3rd party software.

The libraries STL (shipped with any compiler) and BOOST are essential to all components (i.e. libCGAL, libCGAL_Core, libCGAL_imageIO, libCGAL_Qt3 and libCGAL_Qt4).

3.8.1 Standard Template Library (STL)

CGAL heavily uses the STL, and in particular adopted many of its design ideas. You can find online documentation for the STL at various web sites, for instance, `http://www.sgi.com/tech/stl/`, `http://www.cplusplus.com/reference/`, or `http://msdn.microsoft.com/en-us/library/1fe2x6kt(VS.100).aspx`.

[15] CGAL_Core is not part of CGAL, it is a custom version the CORE library distributed by CGAL for the user convenience and it has it's own license.

The STL comes with the compiler, so there is nothing to install.

3.8.2 BOOST

The BOOST libraries are a set of portable C++ source libraries. Most of BOOST libraries are header-only, but a few of them need to be compiled or installed as binaries.

CGAL requires the BOOST libraries. In particular the header files and the threading library (`Boost.Thread` and `Boost.System` binaries). Version 1.39 (or higher) are needed.

On Windows, as auto-linking is used, you also need the binaries of `Boost.Serialization` and `Boost.DateTime`, but the dependency is artificial and used only at link-time: the CGAL libraries do not depend on the DLL's of those two libraries.

In CGAL some demos and examples depend on `Boost.Program_options`.

In case the BOOST libraries are not installed on your system already, you can obtain them from `http://www.boost.org/`. For Windows you can download an installer from `http://www.boostpro.com/download/`. Since `Boost.Thread` is required, make sure to either install the precompiled libraries for your compiler or build `libboost-thread` and `libboost-system`.

As on Windows there is no canonical directory for where to find BOOST, we recommend that you define the environment variable *BOOST_ROOT* and set it to where you have installed BOOST, e.g., `C:\boost\boost_1_41_0`.

3.8.3 GMP and MPFR

The components libCGAL, libCGAL_Core, libCGAL_Qt3 and libCGAL_Qt4 require GMP and MPFR which are libraries for multi precision integers and rational numbers, and for multi precision floating point numbers.

CGAL combines floating point arithmetic with exact arithmetic, in order to be efficient and reliable. CGAL has a built-in number type for that, but GMP and MPFR provide a faster solution, and we recommend to use them.

Having GMP version 4.2 or higher and MPFR version 2.2.1 or higher installed is recommended. These libraries can be obtained from `http://gmplib.org/` and `http://www.mpfr.org/`, respectively.

As Visual C++ is not properly supported by the GMP and MPFR projects, we provide precompiled versions of GMP and MPFR, which can be downloaded with the installer `CGAL-4.1-Setup.exe`.

3.8.4 ZLIB

ZLIB is a data compression library, and is essential for the component libCGAL_ImageIO.

In CGAL this library is used in the examples of the Surface Mesh Generation (Page **??**) package.

If it is not already on your system, for instance, on Windows, you can download it from `http://www.zlib.net/`.

3.8.5 OPENGL

OPENGL (Open Graphics Library) provides an API for applications that produce 2D and 3D computer graphics.

In CGAL the library is essential for the components libCGAL_Qt3 and libCGAL_Qt4, as well as libCGAL_ImageIO and for various demos.

Typically, OPENGL is preinstalled on systems; if not, it can be downloaded from `http://www.opengl.org/`.

3.8.6 QT3 and QT4

Qt is a cross-platform application and UI framework.

The component libCGAL_Qt3 requires QT3 installed on your system, while the component libCGAL_Qt4 requires QT4 installed on your system. In case QT is not yet installed on your system, you can download it from `http://qt.nokia.com/`.

Older demos of CGALuse libCGAL_Qt3 and QT3, while newer and newly developed demos require libCGAL_Qt4 and QT4.

Having QT4 version 4.3.0 or higher is recommended.

3.9 CGAL Examples and Demos

CGAL is distributed with a large collection of examples and demos. By default, these are **not** configured along with the CGAL libraries, unless you set the variables `WITH_examples=ON` and/or `WITH_demos=ON`.

Nevertheless, even when configured with CGAL, they are not automatically built along with the libraries. You must build the `examples` or `demos` targets (or IDE projects) explicitly.

If you do not plan to compile any demos, you might skip some of the essential libraries (as QT or OPENGL), as the corresponding CGAL-libraries are not linked. But for your own demos you might need these CGAL-libraries.

3.10 Optional Third Party Libraries

Optional 3rd party software can be used by CGAL for various reasons: Usually certain optional libraries are required to build examples and demos shipped with CGAL or to build your own project using CGAL. Another reason is to speed up basic tasks. In order to support these goals, all optional libraries can be *prepared to be used with* CGAL while configuring CGAL, just in the same way as essential libraries are configured. Whenever building an example or a demo (or your own executable), these *preconfigured* libraries are available when using CGAL.

3.10.1 LEDA

LEDA is a library of efficient data structures and algorithms. Like CORE, LEDA offers a real number data type.

In CGAL this library is optional, and its number types can be used as an alternative to GMP, MPFR, and CORE.

Free and commercial editions of LEDA are available from http://www.algorithmic-solutions.com/leda/index.html.

3.10.2 MPFI

MPFI provide arbitrary precision interval arithmetic with intervals represented using MPFR reliable floating-point numbers. It is based on the GMP library and on the MPFR library. In the setting of CGAL it is mainly used in sync with Rs. The library is optional and needed in certain algebraic kernels.

MPFI can be downloaded from http://mpfi.gforge.inria.fr/. Version 1.4 or higher is recommended.

3.10.3 Rs and Rs3

Rs (Real Solutions) is devoted to the study of the real roots of polynomial systems with a finite number of complex roots (including univariate polynomials).

In CGAL, Rs is used by one model of the Algebraic Kernel (Page **??**).

Rs is freely distributable for non-commercial use. You can download it from http://vegas.loria.fr/rs/. The library Rs needs MPFI, which can be downloaded from http://mpfi.gforge.inria.fr/. Version 1.4 or higher is recommended.

The successor of Rs is called Rs3. It less restrictive when it comes to licencing and also contains improved and more efficient interfaces. Mainly parts in CGAL's algebraic kernel require Rs3.

3.10.4 NTL

NTL provides data structures and algorithms for signed, arbitrary length integers, and for vectors, matrices, and polynomials over the integers and over finite fields. The optional library NTL is used by CGAL to speed up operations of the Polynomial package, such as GCDs. It is recommended to install NTL with support from GMP.

NTL can be downloaded from http://www.shoup.net/ntl/. Version 5.1 or higher is recommended.

3.10.5 EIGEN

EIGEN is a C++ template library for linear algebra. EIGEN supports all matrix sizes, various matrix decomposition methods and sparse linear solvers.

In CGAL, EIGEN provides sparse linear solvers in the Surface Reconstruction from Point Sets (Page **??**) and the Planar Parameterization of Triangulated Surface Meshes (Page **??**) packages.

In addition, EIGEN also provides singular value decomposition for the Estimation of Local Differential Properties (Page **??**) and the Approximation of Ridges and Umbilics (Page **??**) packages.

The EIGEN web site is http://eigen.tuxfamily.org.

3.10.6 libQGLViewer

libQGLViewer is a 3D widget based on QT 4's *QGLWidget*.

In CGAL some 3D demos are based on libQGLViewer.

It can be downloaded from `http://www.libqglviewer.com/`.

3.10.7 COIN3D

COIN3D is an implementation of Open Inventor.

In CGAL, COIN3D is used in the demo of the Kinetic Data Structures (Page **??**) package.

You can download it from `http://www.coin3d.org/`.

3.10.8 ESTBL

The ESTBL (Easy Structural Biology Template Library) is a library that allows the handling of PDB data.

In CGAL the ESTBL is used in an example of the 3D Skin Surface Meshing (Page **??**) package.

It can be downloaded from `http://esbtl.sourceforge.net/`.

3.11 Building CGAL

The results of a successful configuration are build files that control the build step. The nature of the build files depends on the generator used during configuration, but in all cases they contain several *targets*, one per library, and a default global target corresponding to all the libraries.

For example, in a UNIX-like environment the default generator produces makefiles. You can use the `make` command-line tool for the succeeding build step as follows:

```
cd CGAL-4.1

# build all the selected libraries at once

make
```

The resulting libraries are placed in the subdirectory `lib` under <CMAKE_BINARY_DIR> (which is `CGAL-4.1` in case you run an in-source-configuration).

With generators other than `UNIX Makefiles` the resulting build files are solution and project files which should be launched in an IDE, such as Visual Studio or KDevelop3. They will contain the targets described above, which you can manually build as with any other solution/project within your IDE.

Alternatively, you can build it with the command line version of the VISUAL STUDIO IDE:

```
devenv CGAL.sln /Build Debug
```

The "Debug" argument is needed because CMake creates solution files for all four configurations, and you need to explicitly choose one when building (the other choices are Release, RelWithDebInfo and MinSizeRel).

advanced

The build files produced by CMake are autoconfigured. That is, if you change any of the dependencies, the build step automatically goes all the way back to the configuration step. This way, once the target has been configured the very first time by invoking cmake, you don't necessarily need to invoke `cmake` again. Rebuilding will call itself `cmake` and re-generate the build file whenever needed. Keep this in mind if you configure CGAL for the Visual Studio IDE since a build could then change the solution/project file in-place and VS will prompt you to reload it.

advanced

If you have turned on the configuration of examples (`-DWITH_examples=ON`) and/or demos (`-DWITH_demos=ON`), there will be additional targets named `examples` and `demos`, plus one target for each example and each demo in the build files. None of these targets are included by default, so you need to build them explicitly *after* the CGAL libraries have been successfully built. The targets `examples` and `demos` include themselves all the targets for examples and demos respectively.

```
# build all examples at once
make examples

# build all demos at once
make demos

# build only the Straight Skeleton demo
make Straight_skeleton_2_demo
```

advanced

When using `UNIX Makefiles` you can find out the exact name of the example or demo target of a particular package by typing `make help | grep <package>`.

advanced

3.12 Installing CGAL

On many platforms, library pieces such as headers, docs and binaries are expected to be placed in specific locations. A typical example being `/usr/include` and `/usr/lib` on UNIX-like operating systems or `C:/Program Files/` on Windows. The process of placing or copying the library elements into its standard location is sometimes referred to as *Installation* and it is a postprocessing step after the build step.

CMake carries out the installation by producing a build target named *install*. The following example shows a typical session from configuration to installation in a UNIX-like environment:

```
cd CGAL-4.1

cmake .      # configure
make         # compile
make install # install
```

If you use a generator that produces IDE files (for Visual Studio for instance) there will be an optional `INSTALL` project, which you will be able to *"build"* to execute the installation step.

advanced

The files are copied into a directory tree relative to the *installation directory* determined by the CMake variable `CMAKE_INSTALL_PREFIX`. This variable defaults to `/usr/local` under UNIX-like operating systems and `C:\Program Files` under Windows. If you want to install to a different location, you must override that CMake variable explicitly *at the configuration time* and not when executing the install step.

advanced

The file `CGALConfig.cmake` is installed by default in `$CMAKE_INSTALLED_PREFIX/lib/CGAL-4.1`.

3.13 Example Configuration

Below is an example output on a linux machine with g++4.4 installed, using CMake 2.8.5, and the following command-line call to `cmake`:

```
cmake -DWITH_examples=OFF -DWITH_demos=OFF -DCMAKE_BUILD_TYPE=Release
  /path/to/unpacked/CGAL-tarball/
```

```
-- The CXX compiler identification is GNU
-- The C compiler identification is GNU
-- Check for working CXX compiler: /usr/bin/g++-4.4
-- Check for working CXX compiler: /usr/bin/g++-4.4 -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Check for working C compiler: /usr/bin/gcc-4.4
-- Check for working C compiler: /usr/bin/gcc-4.4 -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
== Setting paths ==
== Build CGAL from release: CGAL-4.1 ==
-- Packagenames: CGAL-4.1
== Setting paths (DONE) ==

== Generate version files ==
-- CGAL_MAJOR_VERSION=4
```

```
-- CGAL_MINOR_VERSION=1
-- CGAL_BUGFIX_VERSION=0
-- CGAL_SONAME_VERSION=10
-- CGAL_SOVERSION     =10.0.0
-- CGAL_REFERENCE_CACHE_DIR=
-- Building shared libraries
-- Targetting Unix Makefiles
-- Using /usr/bin/g++-4.4 compiler.
-- USING CMake version: 2.8.5
-- System: Linux
-- USING GCC_VERSION = '4.4.5'
-- Using gcc version 4 or later. Adding -frounding-math
== Generate version files (DONE) ==

== Detect external libraries ==
-- Build type: Release
-- USING CXXFLAGS = ' -frounding-math -O3 -DNDEBUG'
-- USING LDFLAGS = '  '
-- External libraries supported: GMP;GMPXX;MPFR;zlib;OpenGL;LEDA;MPFI;RS;RS3;OpenNL;TAUCS;EIGEN3;BLAS;LAP
-- Preconfiguring library: GMP ...
-- GMP has been preconfigured:
--   CGAL_UseGMP-file:
--   GMP include:      /usr/include/
--   GMP libraries:    /usr/lib/libgmp.so
--   GMP definitions:
-- USING GMP_VERSION = '4.3.2'
-- Preconfiguring library: GMPXX ...
-- GMPXX has been preconfigured:
--   CGAL_UseGMPXX-file:
--   GMPXX include:      /usr/include
--   GMPXX libraries:    /usr/lib/libgmpxx.so
--   GMPXX definitions:
-- Preconfiguring library: MPFR ...
-- MPFR has been preconfigured:
--   CGAL_UseMPFR-file:
--   MPFR include:      /usr/include/
--   MPFR libraries:    /usr/lib/libmpfr.so
--   MPFR definitions:
-- USING MPFR_VERSION = '3.0.0'
-- Boost version: 1.39.0
-- Found the following Boost libraries:
--   thread
-- Boost include:     /usr/include/boost_1_39_0/include/boost-1_39
-- Boost libraries:   /usr/lib/libboost_thread-mt.so;pthread
-- Boost definitions:
-- USING BOOST_VERSION = '1.39.0'
== Detect external libraries (DONE) ==

== Write compiler_config.h ==
-- Performing Test CGAL_CFG_ARRAY_MEMBER_INITIALIZATION_BUG - Success
-- Performing Test CGAL_CFG_DENORMALS_COMPILE_BUG - Success
-- Performing Test CGAL_CFG_FPU_ROUNDING_MODE_UNWINDING_VC_BUG - Success
-- Performing Test CGAL_CFG_IEEE_754_BUG - Success
-- Performing Test CGAL_CFG_ISTREAM_INT_BUG - Success
```

```
-- Performing Test CGAL_CFG_LONGNAME_BUG - Success
-- Performing Test CGAL_CFG_MATCHING_BUG_5 - Success
-- Performing Test CGAL_CFG_MATCHING_BUG_6 - Success
-- Performing Test CGAL_CFG_NESTED_CLASS_FRIEND_DECLARATION_BUG - Success
-- Performing Test CGAL_CFG_NO_CPP0X_ARRAY - Failed
-- Performing Test CGAL_CFG_NO_CPP0X_AUTO - Failed
-- Performing Test CGAL_CFG_NO_CPP0X_COPY_N - Failed
-- Performing Test CGAL_CFG_NO_CPP0X_DECLTYPE - Failed
-- Performing Test CGAL_CFG_NO_CPP0X_DEFAULT_TEMPLATE_ARGUMENTS_FOR_FUNCTION_TEMPLATES - Failed
-- Performing Test CGAL_CFG_NO_CPP0X_DELEGATING_CONSTRUCTORS - Failed
-- Performing Test CGAL_CFG_NO_CPP0X_DELETED_AND_DEFAULT_FUNCTIONS - Success
-- Performing Test CGAL_CFG_NO_CPP0X_INITIALIZER_LISTS - Failed
-- Performing Test CGAL_CFG_NO_CPP0X_ISFINITE - Success
-- Performing Test CGAL_CFG_NO_CPP0X_LAMBDAS - Failed
-- Performing Test CGAL_CFG_NO_CPP0X_LONG_LONG - Success
-- Performing Test CGAL_CFG_NO_CPP0X_NEXT_PREV - Failed
-- Performing Test CGAL_CFG_NO_CPP0X_RVALUE_REFERENCE - Failed
-- Performing Test CGAL_CFG_NO_CPP0X_STATIC_ASSERT - Failed
-- Performing Test CGAL_CFG_NO_CPP0X_TUPLE - Failed
-- Performing Test CGAL_CFG_NO_CPP0X_VARIADIC_TEMPLATES - Failed
-- Performing Test CGAL_CFG_NO_LIMITS - Success
-- Performing Test CGAL_CFG_NO_LOGICAL_OPERATORS_ALTERNATIVES - Success
-- Performing Test CGAL_CFG_NO_MESSAGE_PRAGMA_BUG - Success
-- Performing Test CGAL_CFG_NO_NEXTAFTER - Success
-- Performing Test CGAL_CFG_NO_STATEMENT_EXPRESSIONS - Success
-- Performing Test CGAL_CFG_NO_STL - Success
-- Performing Test CGAL_CFG_NO_TR1_ARRAY - Success
-- Performing Test CGAL_CFG_NO_TR1_TUPLE - Success
-- Performing Test CGAL_CFG_NO_WARNING_CPP_DIRECTIVE_BUG - Success
-- Performing Test CGAL_CFG_NUMERIC_LIMITS_BUG - Success
-- Performing Test CGAL_CFG_OUTOFLINE_MEMBER_DEFINITION_BUG - Success
-- Performing Test CGAL_CFG_TEMPLATE_IN_DEFAULT_PARAMETER_BUG - Success
-- Performing Test CGAL_CFG_TYPENAME_BEFORE_DEFAULT_ARGUMENT_BUG - Success
-- Performing Test CGAL_CFG_USING_BASE_MEMBER_BUG_2 - Success
== Write compiler_config.h (DONE) ==

== Generating build files ==
-- Configure libCGAL
-- GMP include:     /usr/include/
-- GMP definitions:
-- GMP libraries:   /usr/lib/libgmp.so
-- Configured GMP in standard way
-- MPFR include:     /usr/include/
-- MPFR definitions:
-- MPFR libraries:   /usr/lib/libmpfr.so
-- Configured MPFR in standard way
-- libCGAL is configured
-- Sources for CGAL component library 'Core' detected
-- Configure libCGALCore
-- GMP include:     /usr/include/
-- GMP definitions:
-- GMP libraries:   /usr/lib/libgmp.so
-- Configured GMP in standard way
-- MPFR include:     /usr/include/
```

```
-- MPFR definitions:
-- MPFR libraries:   /usr/lib/libmpfr.so
-- Configured MPFR in standard way
-- libCGALCore is configured
-- Sources for CGAL component library 'Qt3' detected
-- Sources for CGAL component library 'Qt4' detected
-- Configure libCGALQt4
-- Looking for Q_WS_X11
-- Looking for Q_WS_X11 - found
-- Looking for Q_WS_WIN
-- Looking for Q_WS_WIN - not found.
-- Looking for Q_WS_QWS
-- Looking for Q_WS_QWS - not found.
-- Looking for Q_WS_MAC
-- Looking for Q_WS_MAC - not found.
-- Looking for XOpenDisplay in /usr/lib/libX11.so;/usr/lib/libXext.so
-- Looking for XOpenDisplay in /usr/lib/libX11.so;/usr/lib/libXext.so - found
-- Looking for gethostbyname
-- Looking for gethostbyname - found
-- Looking for connect
-- Looking for connect - found
-- Looking for remove
-- Looking for remove - found
-- Looking for shmat
-- Looking for shmat - found
-- Looking for IceConnectionNumber in ICE
-- Looking for IceConnectionNumber in ICE - found
-- Found X11: /usr/lib/libX11.so
-- USING QT4_VERSION = '4.6.3'
-- OpenGL include:     /usr/include
-- OpenGL libraries:   /usr/lib/libGLU.so;/usr/lib/libGL.so;/usr/lib/libSM.so;/usr/lib/libICE.so;/usr/lib
-- OpenGL definitions:
-- Qt4 include:        /usr/include/qt4
-- Qt4 libraries:      /usr/lib/libQtOpenGL.so;/usr/lib/libQtGui.so;/usr/lib/libQtCore.so
-- Qt4 definitions:
-- moc executable:     /usr/bin/moc-qt4
-- uic executable:     /usr/bin/uic-qt4
-- GMP include:     /usr/include/
-- GMP definitions:
-- GMP libraries:   /usr/lib/libgmp.so
-- Configured GMP in standard way
-- MPFR include:     /usr/include/
-- MPFR definitions:
-- MPFR libraries:   /usr/lib/libmpfr.so
-- Configured MPFR in standard way
-- libCGALQt4 is configured
-- Sources for CGAL component library 'ImageIO' detected
-- Configure libCGALImageIO
-- Found OpenGL: /usr/lib/libGL.so
-- Found ZLIB: /usr/lib/libz.so (found version "1.2.3.4")
-- OpenGL include:     /usr/include
-- OpenGL libraries:   /usr/lib/libGLU.so;/usr/lib/libGL.so;/usr/lib/libSM.so;/usr/lib/libICE.so;/usr/lib
-- USING ZLIB_VERSION = '1.2.3.4'
-- libCGALImageIO is configured
```

```
-- Sources for CGAL component libraries 'CGAL;Core;ImageIO;Qt3;Qt4' detected
== Generating build files (DONE) ==

-- Configuring done
-- Generating done
-- Build files have been written to: /home/user/CGAL/4.1/
```

3.14 Multiple Variants of makefiles (out-of-source build)

While you can choose between release or debug builds, and shared or static libraries, it is not possible to generate different variants during a single configuration. You need to run CMake in a different directory for each variant you are interested in, each with its own selection of configuration parameters.

CMake stores the resulting makefiles and project files, along with several temporary and auxiliary files such as the variables cache, in the directory where it is executed, called `CMAKE_BINARY_DIR`, but it takes the source files and configuration scripts from `CMAKE_SOURCE_DIR`.

The binary and source directories do not need to be the same. Thus, you can configure multiple variants by creating a distinct directory for each configuration and by running CMake from there. This is known in CMake terminology as *out-of-source configuration*, as opposite to an *in-source configuration*, as showed in the previous sections.

You can, for example, generate subdirectories `CGAL-4.1/cmake/platforms/debug` and `CGAL-4.1/cmake/platforms/release` for two configurations, respectively:

```
mkdir CGAL-4.1/cmake/platforms/debug
cd CGAL-4.1/cmake/platforms/debug
cmake -DCMAKE_BUILD_TYPE=Debug ../../..

mkdir CGAL-4.1/cmake/platforms/release
cd CGAL-4.1/cmake/platforms/release
cmake -DCMAKE_BUILD_TYPE=Release ../../..
```

3.15 Configuring and Building Programs Using CGAL

Ideally, configuring and compiling a demo/example/program amounts to

```
cd CGAL-4.1/examples/Straight_skeleton_2
cmake -DCGAL_DIR=$HOME/CGAL-4.1 .
make
```

In this ideal world, as for all shipped examples and demos of CGAL, the required `CMakeLists.txt` is already provided.

CMake can also be used to configure and build user programs via such CMake-scripts. In this less ideal world, one has to provide the `CMakeLists.txt` script either manually, or with the help of a shell-script that is introduced below.

For a user program `executable.cpp`, the ideal world looks like this:

```
cd /path/to/program
cgal_create_CMakeLists -s executable
cmake -DCGAL_DIR=$HOME/CGAL-4.1 .
make
```

In both examples we specify the `CGAL_DIR`: During configuration of the CGAL libraries a file named `CGALConfig.cmake` is generated in CGAL's root directory (in contrast to CGAL's source directory that has been used for installation). This file contains the definitions of several CMake variable that summarize the configuration of CGAL. In order to configure a program, you need to indicate the location of that config file in the CMake variable `CGAL_DIR` (as indicated in the example above). `CGAL_DIR` can also be an environment variable. Setting `CGAL_DIR` makes particular sense if having multiple out-of-source builds of CGAL as in Section 3.14.

If you have installed CGAL, `CGAL_DIR` must afterwards be set to `$CMAKE_INSTALLED_PREFIX/lib/CGAL`. Note that CGAL is recommended to be installed in release mode when using it to build programs.

3.15.1 Creating a cmake script for a program using CGAL

For compiling a non-shipped program, it is recommended, to also rely on a CMake-supported configuration using a `CMakeLists.txt` used for configuration.

Use the following Bourne-shell script for programs that are relatively simple to configure:

cgal_create_CMakeLists

The Bourne-shell script `cgal_create_CMakeLists.txt` resides in the `CGAL-4.1/scripts` directory. It can be used to create `CMakeLists.txt` files for compiling CGAL applications. Executing `cgal_create_CMakeLists.txt` in an application directory creates a `CMakeLists.txt` containing rules to build the contained application(s). Three command line options determine details of the configuration.

-s source If this parameter is given the script will create **a single executable** for 'source' linked with compilations of all other source files (`*.cpp`). This behaviour is usually needed for (graphical) demos.
If the parameter is not given, the script creates **one executable for each given source file**.

-c com1:com2:... Lists components ("com1", "com2") of CGAL to which the executable(s) should be linked. Valid components are CGAL's libraries (i.e. "Core", "ImageIO", "Qt3" and "Qt4"; note that it only make sense to either pick "Qt3" or "Qt4") and all preconfigured 3rd party software, such as "MPFI" or "RS3"). An example is `-c Core:GMP:RS3:MPFI`

-b boost1:boost2:... Lists components ("boost1", "boost2") of BOOST to which the executable(s) should be linked. Valid options are, for instance, "filesystem" or "program_options".

This options should suffice to create `CMakeLists.txt` script for most directories containing programs. However, in some special cases, it might still be required to create the script manually, for instance, if some source files/executables need a different linking than other source files.

deprecated

cgal_create_cmake_script

For backward-compatibility we still provide the Bourne-shell script `cgal_create_cmake_script` that is contained in the `CGAL-4.1/scripts` directory. It can be used to create `CMakeLists.txt` files for compiling CGAL applications. Executing `cgal_create_cmake_script` in an application directory creates a `CMakeLists.txt` containing rules for every `*.cpp` file there. The script is deprecated, as it only works for applications with a single course file that only need libCGAL and libCGAL_Core.

deprecated

Such a shell-script simply creates a CMake script. Processing it with CMake, searches for CGAL using `find_package`. If found, the variable `CGAL_USE_FILE` is set to a compilation environment CMake file. Including this file within a CMake script sets up include paths and libraries to link with CGAL and essential third party libraries. Beyond, `find_package` can demand for `COMPONENTS` of CGAL, that is, all CGAL libraries libCGAL_Core ("Core"), libCGAL_imageIO ("ImageIO") , libCGAL_Qt3 ("Qt3") and libCGAL_Qt4 ("Qt4") or optional 3rd party software such as "MPFI" or "RS3". A user is free to create the `CMakeLists.txt` without calling the script (manual creation).

3.15.2 Custom flags in the programs using CGAL

Normally, programs linked with CGAL must be compiled with the same flags used by the compilation of CGAL libraries. For this reason, the *very first* time a program is configured, all the flags given by the CMake variables `CMAKE_*_FLAGS` are *locked* in the sense that the values recorded in `CGALConfig.cmake` are used to override any values given by CMake itself or yourself.

This does not apply to the additional flags that can be given via `CGAL_*_FLAGS`.

Such *inherited* values are then recorded in the current CMake cache for the program. The flags are then *unlocked* in the sense that at any subsequent configuration you can provide your own flags and this time they will not be overridden.

When using the interactive `cmake-gui` the first press on `Configure` unlocks the flags, so that you can edit them as needed.

advanced

The locking of flags is controlled by the variable `CGAL_DONT_OVERRIDE_CMAKE_FLAGS` which starts out FALSE and is toggled right after the flags have been loaded from `CGALConfig.cmake`.

If you use the command line tool you can specify flags *directly* by setting the controlling variable right up front:

```
cd CGAL-4.1

cmake -DCMAKE_BUILD_TYPE=Release -DCMAKE_CXX_FLAGS=-g .

cd CGAL-4.1/examples/Straight_skeleton_2

cmake -DCGAL_DIR=CGAL-4.1 -DCMAKE_BUILD_TYPE=Debug -DCMAKE_CXX_FLAGS=-O2 -DCGAL_DONT_OVERRIDE_CMAKE_FLAGS
```

3.16 Summary of CGAL's Configuration Variables

Most configuration variables are not environment variables but *CMake variables*. They are given in the command line to CMake via the `-D` option, or passed from the interactive interface of `cmake-gui`. Unless indicated differently, all the variables summarized below are CMake variables.

3.16.1 Component selection

The following boolean variables indicate which CGAL components to configure and build. Their values can be ON or OFF.

Variable	Default value
`WITH_examples`	OFF
`WITH_demos`	OFF
`WITH_CGAL_Core`	ON
`WITH_CGAL_Qt3`	ON
`WITH_CGAL_Qt4`	ON
`WITH_CGAL_ImageIO`	ON

3.16.2 Compiler and Linker Flags

The following variables specify compiler and linker flags. Each variable holds a space-separated list of command-line switches for the compiler and linker and their default values are automatically defined by CMake based on the target platform.

Have in mind that these variables specify a list of flags, not just one single flag. If you provide your own definition for a variable, you will *entirely* override the list of flags chosen by CMake for that particular variable.

The variables that correspond to both debug and release builds are always used in conjunction with those for the specific build type.

Program	Both Debug and Release	Release only	Debug Only
C++ Compiler	`CMAKE_CXX_FLAGS`	`CMAKE_CXX_FLAGS_RELEASE`	`CMAKE_CXX_FLAGS_DEBUG`
Linker (shared libs)	`CMAKE_SHARED_LINKER_FLAGS`	`CMAKE_SHARED_LINKER_FLAGS_RELEASE`	`CMAKE_SHARED_LINKER_FLAGS_DEBUG`
Linker (static libs)	`CMAKE_MODULE_LINKER_FLAGS`	`CMAKE_MODULE_LINKER_FLAGS_RELEASE`	`CMAKE_MODULE_LINKER_FLAGS_DEBUG`
Linker (programs)	`CMAKE_EXE_LINKER_FLAGS`	`CMAKE_EXE_LINKER_FLAGS_RELEASE`	`CMAKE_EXE_LINKER_FLAGS_DEBUG`

Additional Compiler and Linker Flags

The following variables can be used to *add* flags without overriding the ones defined by cmake.

Program	Both Debug and Release	Release only	Debug Only
C++ Compiler	`CGAL_CXX_FLAGS`	`CGAL_CXX_FLAGS_RELEASE`	`CGAL_CXX_FLAGS_DEBUG`
Linker (shared libs)	`CGAL_SHARED_LINKER_FLAGS`	`CGAL_SHARED_LINKER_FLAGS_RELEASE`	`CGAL_SHARED_LINKER_FLAGS_DEBUG`
Linker (static libs)	`CGAL_MODULE_LINKER_FLAGS`	`CGAL_MODULE_LINKER_FLAGS_RELEASE`	`CGAL_MODULE_LINKER_FLAGS_DEBUG`
Linker (programs)	`CGAL_EXE_LINKER_FLAGS`	`CGAL_EXE_LINKER_FLAGS_RELEASE`	`CGAL_EXE_LINKER_FLAGS_DEBUG`

3.16.3 Miscellaneous Variables

Variable	Description	Type	Default value
`CMAKE_BUILD_TYPE`	Indicates type of build. Possible values are 'Debug' or 'Release'	CMake	Release
`CMAKE_CXX_COMPILER`	Full-path to the executable corresponding to the C++ compiler to use.	CMake	platform-dependent
`CXX`	Idem	Environment	Idem

Variables used only when building programs (such as demos or examples)

Variable	Description	Type	Default value
`CGAL_DIR`	Full-path to the binary directory where CGAL was configured	Either CMake or Environment	*none*

3.16.4 Variables providing information about 3rd-party libraries

The following variables provide information about the availability and location of the 3rd party libraries used by CGAL. CMake automatically searches for dependencies so you need to specify these variables if CMake was unable to locate something. This is indicated by a value ending in `NOTFOUND`.

Since 3rd-party libraries are system wide, many of the CMake variables listed below can alternatively be given as similarly-named environment variables instead. Keep in mind that you must provide one or the other but never both.

BOOST libraries

In most cases, if BOOST is not automatically found, setting the `BOOST_ROOT` variable is enough. If it is not, you can specify the header and library directories individually. You can also provide the full pathname to a specific compiled library if it cannot be found in the library directory or its name is non-standard.

By default, when BOOST binary libraries are needed, the shared versions are used if present. You can set the variable `CGAL_Boost_USE_STATIC_LIBS` to `ON` if you want to link with static versions explicitly.

On Windows, if you link with BOOST shared libraries, you must ensure that the `.dll` files are found by the dynamic linker, at run time. For example, you can add the path to the BOOST `.dll` to the `PATH` environment variable.

Variable	Description	Type
`BOOST_ROOT`[16]	Root directory of your BOOST installation	Either CMake or Environment
`Boost_INCLUDE_DIR`	Directory containing the `boost/version.hpp` file	CMake
`BOOST_INCLUDEDIR`	Idem	Environment
`Boost_LIBRARY_DIRS`	Directory containing the compiled BOOST libraries	CMake
`BOOST_LIBRARYDIR`	Idem	Environment
`Boost_(xyz)_LIBRARY_RELEASE`	Full pathname to a release build of the compiled 'xyz' BOOST library	CMake
`Boost_(xyz)_LIBRARY_DEBUG`	Full pathname to a debug build of the compiled 'xyz' BOOST library	CMake

GMP and MPFR libraries

Under Windows, auto-linking is used, so only the *directory* containing the libraries is needed and you would specify `GMP|MPFR_LIBRARY_DIR` rather than `GMP|MPFR_LIBRARIES`. On the other hand, under Linux the actual library filename is needed. Thus you would specify `GMP|MPFR_LIBRARIES`. In no case you need to specify both.

CGAL uses both GMP and MPFR so both need to be supported. If either of them is unavailable the usage of GMP and of MPFR will be disabled.

Variable	Description	Type
`WITH_GMP`	Indicates whether to search and use GMPMPFR or not	CMake
`GMP_DIR`	Directory of GMP default installation	Environment
`GMP_INCLUDE_DIR`	Directory containing the `gmp.h` file	CMake
`GMP_INC_DIR`	Idem	Environment
`GMP_LIBRARIES_DIR`	Directory containing the compiled GMP library	CMake
`GMP_LIB_DIR`	Idem	Environment
`GMP_LIBRARIES`	Full pathname of the compiled GMP library	CMake
`MPFR_INCLUDE_DIR`	Directory containing the `mpfr.h` file	CMake
`MPFR_INC_DIR`	Idem	Environment
`MPFR_LIBRARIES_DIR`	Directory containing the compiled MPFR library	CMake
`MPFR_LIB_DIR`	Idem	Environment
`MPFR_LIBRARIES`	Full pathname of the compiled MPFR library	CMake

Under Linux, the GMPXX is also searched for, and you may specify the following variables:

Variable	Description	Type
`GMPXX_DIR`	Directory of GMPXX default installation	Environment
`GMPXX_INCLUDE_DIR`	Directory containing the `gmpxx.h` file	CMake
`GMPXX_LIBRARIES`	Full pathname of the compiled GMPXX library	CMake

QT3 library

In most cases, if QT3 is not automatically found, setting the `QTDIR` environment variable is sufficient. If it is not, you can specify the directory containing the header files and the full pathnames of the QT3 libraries.

[16] The environment variable can be spelled either `BOOST_ROOT` or `BOOSTROOT`

Variable	Description	Type
QTDIR	Root directory of the QT3 library	Environment
QT3_INCLUDE_DIR	Directory containing the qt.h file	CMake
QT3_QT_LIBRARY	Full pathname to the qt library of QT3	CMake
QT3_QTMAIN_LIBRARY	Full pathname to the qtmain library of QT3	CMake
QT3_QASSISTANTCLIENT_LIBRARY	Full pathname to the qassistantclient library of QT3	CMake
QT3_MOC_EXECUTABLE	Full pathname to the moc executable of QT3	CMake
QT3_UIC_EXECUTABLE	Full pathname to the uic executable of QT3	CMake

QT4 library

The CMake scripts that search for QT4 can use the introspection feature of the tool `qmake` included in QT4 distributions. If QT4 is not automatically found, it is sufficient to set the `PATH` environment variable, so that QT4 `qmake` tool is in the path, and before QT3 `qmake` if that one exists. One can alternatively set the CMake variable `QT_QMAKE_EXECUTABLE`. The following variables should be then assigned automatically by CMake.

Variable	Description	Type
QT_INCLUDE_DIR	Directory containing the QtCore/qglobal.h file	CMake
QT_LIBRARY_DIR	Directory containing the compiled QT4 libraries	CMake
QT_*(xyz)*_LIBRARY	Full pathname to the compiled 'xyz' QT4 library[17]	CMake
QT_QMAKE_EXECUTABLE	Full pathname to the qmake executable of QT4	CMake
QT_MOC_EXECUTABLE	Full pathname to the moc executable of QT4	CMake
QT_UIC_EXECUTABLE	Full pathname to the uic executable of QT4	CMake

LEDA library

When the LEDA libraries are not automatically found, yet they are installed on the system with base names 'leda' and 'ledaD' (for the release and debug versions resp.), it might be sufficient to just indicate the library directory via the `LEDA_LIBRARY_DIRS` variable. If that doesn't work because, for example, the names are different, you can provide the full pathnames of each variant via `LEDA_LIBRARY_RELEASE` and `LEDA_LIBRARY_DEBUG`.

The variables specifying definitions and flags can be left undefined if they are not needed by LEDA.

[17] If both release and debug versions are available, this variable contains a list of the following form: '`optimized;<fullpath-to-release-lib>;debug;<fullpath-to-debug-lib>`', where the '`optimized`' and '`debug`' tags should appear verbatim.

Variable	Description	Type
WITH_LEDA	Indicates whether to search and use LEDA or not	CMake
LEDA_DIR	Directory of LEDA default installation	Environment
LEDA_INCLUDE_DIR	Directory containing the file LEDA/system/basic.h	CMake
LEDA_LIBRARIES	Directory containing the compiled LEDA libraries	CMake
LEDA_INC_DIR	Directory containing the file LEDA/system/basic.h	Environment
LEDA_LIB_DIR	Directory containing the compiled LEDA libraries	Environment
LEDA_LIBRARY_RELEASE	Full pathname to a release build of the LEDA library	Either CMake
LEDA_LIBRARY_DEBUG	Full pathname to a debug build of the LEDA library	Either CMake
LEDA_DEFINITIONS	Preprocessor definitions	Either CMake
LEDA_CXX_FLAGS	Compiler flags	Either CMake
LEDA_LINKER_FLAGS	Linker flags	Either CMake

MPFI library

CGAL provides a number type based on this library, but the CGAL library itself does not depend on MPFI. This means that this library must be configured when compiling an application that uses the above number type.

When MPFI files are not on the standard path, the locations of the headers and library files must be specified by using environment variables.

Variable	Description	Type
MPFI_DIR	Directory of MPFI default installation	Environment
MPFI_INCLUDE_DIR	Directory containing the mpfi.h file	CMake
MPFI_INC_DIR	Idem	Environment
MPFI_LIBRARIES_DIR	Directory containing the compiled MPFI library	CMake
MPFI_LIB_DIR	Idem	Environment
MPFI_LIBRARIES	Full pathname of the compiled MPFI library	CMake

Rs and Rs3 library

As said before, only the CGAL univariate algebraic kernel depends on the library Rs. As the algebraic kernel is not compiled as a part of the CGAL library, this library is not detected nor configured at installation time.

CMake will try to find Rs in the standard header and library directories. When it is not automatically detected, the locations of the headers and library files must be specified using environment variables.

Rs needs GMP 4.2 or later and MPFI 1.3.4 or later. The variables related to the latter library may also need to be defined.

Variable	Description	Type
RS_DIR	Directory of Rs default installation	Environment
RS_INCLUDE_DIR	Directory containing the rs_exports.h file	CMake
RS_INC_DIR	Idem	Environment
RS_LIBRARIES_DIR	Directory containing the compiled Rs library	CMake
RS_LIB_DIR	Idem	Environment
RS_LIBRARIES	Full pathname of the compiled Rs library	CMake

Similar variables exist for Rs3.

Variable	Description	Type
RS3_DIR	Directory of Rs3 default installation	Environment
RS3_INCLUDE_DIR	Directory containing the rs_exports.h file	CMake
RS3_INC_DIR	Idem	Environment
RS3_LIBRARIES_DIR	Directory containing the compiled Rs library	CMake
RS3_LIB_DIR	Idem	Environment
RS3_LIBRARIES	Full pathname of the compiled Rs library	CMake

NTL library

Some polynomial computations in CGAL's algebraic kernel are speed up when NTL is available. As the algebraic kernel is not compiled as a part of the CGAL library, this library is not detected nor configured at installation time.

CMake will try to find NTL in the standard header and library directories. When it is not automatically detected, the locations of the headers and library files must be specified using environment variables.

Variable	Description	Type
NTL_DIR	Directory of NTL default installation	Environment
NTL_INCLUDE_DIR	Directory containing the NTL/ZZX.h file	CMake
NTL_INC_DIR	Idem	Environment
NTL_LIBRARIES_DIR	Directory containing the compiled NTL library	CMake
NTL_LIB_DIR	Idem	Environment
NTL_LIBRARIES	Full pathname of the compiled NTL library	CMake

EIGEN library

EIGEN is a header-only template library. Only the *directory* containing the header files of EIGEN 3.1 (or greater) is needed.

Variable	Description	Type
EIGEN3_DIR	Directory of EIGEN default installation	Environment
EIGEN3_INCLUDE_DIR	Directory containing the file signature_of_eigen3_matrix_library	CMake
EIGEN3_INC_DIR	Idem	Environment

QGLViewer library

Some demos require the GLViewer library.

In most cases, if QGLViewer is not automatically found, setting the `QGLVIEWERROOT` environment variable is sufficient. If it is not, you can specify the directory containing the header files and the full pathnames of the release and debug libraries

Variable	Description	Type
`QGLVIEWERROOT`	Root directory of the QGLViewer library	Environment
`QGLVIEWER_INCLUDE_DIR`	Directory containing the `QGLViewer/qglviewer.h` file	CMake
`QGLVIEWER_LIBRARY_RELEASE`	Full pathname to a release build of the QGLViewer library	CMake
`QGLVIEWER_LIBRARY_DEBUG`	Full pathname to a debug build of the QGLViewer library	CMake

ESBTL library

One skin surface example requires the ESBTL library in order to read PDB files.

If ESBTL is not automatically found, setting the `ESBTL_INC_DIR` environment variable is sufficient.

Variable	Description	Type
`ESBTL_DIR`	Directory of ESTBL default installation	Environment
`ESBTL_INC_DIR`	Directory containing the `ESBTL/default.h` file	Environment
`ESBTL_INCLUDE_DIR`	Directory containing the `ESBTL/default.h` file	CMake

3.17 Compiler Workarounds

A number of boolean flags are used to workaround compiler bugs and limitations. They all start with the prefix `CGAL_CFG`. These flags are used to work around compiler bugs and limitations. For example, the flag `CGAL_CFG_NO_CPP0X_LONG_LONG` denotes that the compiler does not know the type `long long`.

For each installation a file `<CGAL/compiler_config.h>` is defined, with the correct settings of all flags. This file is generated automatically by CMake, and it is located in the `include` directory of where you run CMake. For an in-source configuration this means `CGAL-x.y/include`.

The test programs used to generate the `compiler_config.h` file can be found in `config/testfiles`. Both `compiler_config.h` and the test programs contain a short description of the problem. In case of trouble with one of the `CGAL_CFG` flags, it is a good idea to take a look at it.

The file `CGAL/compiler_config.h` is included from `<CGAL/config.h>`. which is included by all CGAL header files.

3.18 Compiler Optimizations

By default CMake generates makefiles for Release mode, with optimization flags switched on, and vcproj files for Release and Debug modes.

Installation
Reference Manual

Joachim Reichel and Fernando Cacciola

This chapter contains no Reference Manual pages.

Part II

Arithmetic and Algebra

Chapter 4

Algebraic Foundations

Michael Hemmer

Contents

4.1 Introduction

CGAL is targeting towards exact computation with non-linear objects, in particular objects defined on algebraic curves and surfaces. As a consequence types representing polynomials, algebraic extensions and finite fields play a more important role in related implementations. This package has been introduced to stay abreast of these changes. Since in particular polynomials must be supported by the introduced framework the package avoids the term *number type*. Instead the package distinguishes between the *algebraic structure* of a type and whether a type is embeddable on the real axis, or *real embeddable* for short. Moreover, the package introduces the notion of *interoperable* types which allows an explicit handling of mixed operations.

4.2 Algebraic Structures

The algebraic structure concepts introduced within this section are motivated by their well known counterparts in traditional algebra, but we also had to pay tribute to existing types and their restrictions. To keep the interface minimal, it was not desirable to cover all known algebraic structures, e.g., we did not introduce concepts for such basic structures as *groups* or exceptional structures as *skew fields*.

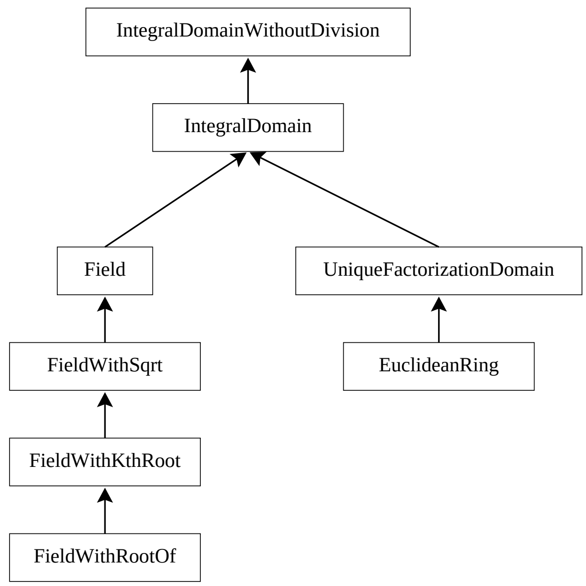


Figure 4.1: Concept Hierarchy of Algebraic Structures

Figure 4.1 shows the refinement relationship of the algebraic structure concepts. *IntegralDomain*, *UniqueFactorizationDomain*, *EuclideanRing* and *Field* correspond to the algebraic structures with the same name. *FieldWithSqrt*, *FieldWithKthRoot* and *FieldWithRootOf* are fields that in addition are closed under the operations 'sqrt', 'k-th root' and 'real root of a polynomial', respectively. The concept *IntegralDomainWithoutDivision* also corresponds to integral domains in the algebraic sense, the distinction results from the fact that some implementations of integral domains lack the (algebraically always well defined) integral division. Note that *Field* refines *IntegralDomain*. This is because most ring-theoretic notions like greatest common divisors become trivial for *Field*s. Hence we see *Field* as a refinement of *IntegralDomain* and not as a refinement of one of the more advanced ring concepts. If an algorithm wants to rely on gcd or remainder computation, it is trying to do things it should not do with a *Field* in the first place.

The main properties of an algebraic structure are collected in the class *Algebraic_structure_traits*. In particular the (most refined) concept each concrete model *AS* fulfills is encoded in the tag *Algebraic_structure_traits<AS> ::Algebraic_category*. An algebraic structure is at least *Assignable*, *CopyConstructible*, *DefaultConstructible* and *EqualityComparable*. Moreover, we require that it is constructible from *int*. For ease of use and since their semantic is sufficiently standard to presume their existence, the usual arithmetic and comparison operators are required to be realized via C++ operator overloading. The division operator is reserved for division in fields. All other unary (e.g., sqrt) and binary functions (e.g., gcd, div) must be models of the well known STL-concepts *AdaptableUnaryFunction* or *AdaptableBinaryFunction* concept and local to the traits class (e.g., *Algebraic_structure_traits<AS>::Sqrt()(x)*). This design allows us to profit from all parts in the STL and its programming style and avoids the name-lookup and two-pass template compilation problems experienced with the old design using overloaded functions. However, for ease of use and backward compatibility all functionality is also accessible through global functions defined within namespace *CGAL*, e.g., *CGAL::sqrt(x)*. This is realized via function templates using the according functor of the traits class. For an overview see Section 4.8 in the reference manual.

4.2.1 Tags in Algebraic Structure Traits

Algebraic Category

For a type *AS*, *Algebraic_structure_traits*<*AS*> provides several tags. The most important tag is the *Algebraic_category* tag, which indicates the most refined algebraic concept the type *AS* fulfills. The tag is one of; *Integral_domain_without_division_tag*, *Integral_domain_tag*, *Field_tag*, *Field_with_sqrt_tag*, *Field_with_kth_root_tag*, *Field_with_root_of_tag*, *Unique_factorization_domain_tag*, *Euclidean_ring_tag*, or even *Null_tag* in case the type is not a model of an algebraic structure concept. The tags are derived from each other such that they reflect the hierarchy of the algebraic structure concept, e.g., *Field_with_sqrt_tag* is derived from *Field_tag*.

Exact and Numerical Sensitive

Moreover, *Algebraic_structure_traits*<*AS*> provides the tags *Is_exact* and *Is_numerical_sensitive*, which are both *Boolean_tags*.
An algebraic structure is considered *exact*, if all operations required by its concept are computed such that a comparison of two algebraic expressions is always correct.
An algebraic structure is considered as *numerically sensitive*, if the performance of the type is sensitive to the condition number of an algorithm. Note that there is really a difference among these two notions, e.g., the fundamental type *int* is not numerical sensitive but considered inexact due to overflow. Conversely, types as *leda_real* or *CORE::Expr* are exact but sensitive to numerical issues due to the internal use of multi precision floating point arithmetic. We expect that *Is_numerical_sensitive* is used for dispatching of algorithms, while *Is_exact* is useful to enable assertions that can be check for exact types only.

Tags are very useful to dispatch between alternative implementations. The following example illustrates a dispatch for *Field*s using overloaded functions. The example only needs two overloads since the algebraic category tags reflect the algebraic structure hierarchy.

```
#include <CGAL/basic.h>
#include <CGAL/IO/io.h>
#include <CGAL/Algebraic_structure_traits.h>

template< typename NT > NT unit_part(const NT& x);
template< typename NT >
NT unit_part_(const NT& x, CGAL::Field_tag);
template< typename NT >
NT unit_part_(const NT& x, CGAL::Integral_domain_without_division_tag);

template< typename NT >
NT unit_part(const NT& x){
    // the unit part of 0 is defined as 1.
    if (x == 0 ) return NT(1);

    typedef CGAL::Algebraic_structure_traits<NT> AST;
    typedef typename AST::Algebraic_category Algebraic_category;
    return unit_part_(x,Algebraic_category());
}

template< typename NT >
NT unit_part_(const NT& x, CGAL::Integral_domain_without_division_tag){
    // For many other types the only units are just -1 and +1.
```

```
    return NT(int(CGAL::sign(x)));
}

template< typename NT >
NT unit_part_(const NT& x, CGAL::Field_tag){
    // For Fields every x != 0 is a unit.
    // Therefore, every x != 0 is its own unit part.
    return x;
}

int main(){
    // Function call for a model of EuclideanRing, i.e. int.
    std::cout<< "int:    unit_part(-3  ): " << unit_part(-3  ) <<
   std::endl;
    // Function call for a model of FieldWithSqrt, i.e. double
    std::cout<< "double: unit_part(-3.0): " << unit_part(-3.0) <<
   std::endl;
    return 0;
}

// Note that this is just an example
// This implementation for unit part won't work for some types, e.g.,
// types that are not RealEmbeddable or types representing structures
   that have
// more units than just -1 and +1. (e.g. MP_Float representing Z[1/2])
// From there Algebraic_structure_traits provides the functor Unit_part.
```

File: examples/Algebraic_foundations/algebraic_structure_dispatch.cpp

4.3 Real Embeddable

Most number types represent some subset of the real numbers. From those types we expect functionality to compute the sign, absolute value or double approximations. In particular we can expect an order on such a type that reflects the order along the real axis. All these properties are gathered in the concept *RealComparable*. The concept is orthogonal to the algebraic structure concepts, i.e., it is possible that a type is a model of *RealEmbeddable* only, since the type may just represent values on the real axis but does not provide any arithmetic operations.

As for algebraic structures this concept is also traits class oriented. The main functionality related to *RealEmbeddable* is gathered in the class *Real_embeddable_traits*. In particular, it porivdes the boolean tag *Is_real_embeddable* indicating whether a type is a model of *RealEmbeddable*. The comparison operators are required to be realized via C++ operator overloading. All unary functions (e.g. *sign*, *to_double*) and binary functions (e.g. *compare*) are models of the STL-concepts *AdaptableUnaryFunction* and *AdaptableBinaryFunction* and are local to *Real_embeddable_traits*.

In case a type is a model of *IntegralDomainWithoutDivision* and *RealEmbeddable* the number represented by an object of this type is the same for arithmetic and comparison. It follows that the ring represented by this type is a superset of the integers and a subset of the real numbers and hence has characteristic zero. In case the type is a model of *Field* and *RealEmbeddable* it is a superset of the rational numbers.

4.4 Real Number Types

Every CGAL *Kernel* comes with two *real number types* (number types embeddable into the real numbers). One of them is a *FieldNumberType*, and the other a *RingNumberType*. The coordinates of the basic kernel objects (points, vectors, etc.) come from one of these types (the *FieldNumberType* in case of Cartesian kernels, and the *RingNumberType* for Homogeneous kernels).

The concept *FieldNumberType* combines the requirements of the concepts *Field* and *RealEmbeddable*, while *RingNumberType* combines *IntegralDomainWithoutDivision* and *RealEmbeddable*. Algebraically, the real number types do not form distinct structures and are therefore not listed in the concept hierarchy of Figure 4.1.

4.5 Interoperability

This section introduces two concepts for interoperability of types, namely *ImplicitInteroperable* and *ExplicitInteroperable*. While *ExplicitInteroperable* is the base concept, we start with *ImplicitInteroperable* since it is the more intuitive one.

In general mixed operations are provided by overloaded operators and functions or just via implicit constructor calls. This level of interoperability is reflected by the concept *ImplicitInteroperable*. However, within template code the result type, or so called coercion type, of a mixed arithmetic operation may be unclear. Therefore, the package introduces *CGAL::Coercion_traits* giving access to the coercion type via *CGAL::Coercion_traits< A,B>::Type* for two interoperable types *A* and *B*.

Some trivial example are *int* and *double* with coercion type double or *CGAL::Gmpz* and *CGAL::Gmpq* with coercion type *CGAL::Gmpq*. However, the coercion type is not necessarily one of the input types, e.g. the coercion type of a polynomial with integer coefficients that is multiplied by a rational type is supposed to be a polynomial with rational coefficients.

CGAL::Coercion_traits is also required to provide a functor *CGAL::Coercion_traits<A,B>::Cast()*, that converts from an input type into the coercion type. This is in fact the core of the more basic concept *ExplicitInteroperable*. *ExplicitInteroperable* has been introduced to cover more complex cases for which it is hard or impossible to guarantee implicit interoperability. Note that this functor can be useful for *ImplicitInteroperable* types as well, since it can be used to void redundant type conversions.

In case two types *A* and *B* are *ExplicitInteroperable* with coercion type *C* they are valid argument types for all binary functors provided by *Algebraic_structure_traits* and *Real_embeddable_traits* of *C*. This is also true for the according global functions.

4.5.1 Examples

The following example illustrates how two write code for *ExplicitInteroperable* types.

```
#include <CGAL/basic.h>
#include <CGAL/Coercion_traits.h>
#include <CGAL/IO/io.h>

// this is an implementation for ExplicitInteroperable types
// the result type is determined via Coercion_traits<A,B>
template <typename A, typename B>
typename CGAL::Coercion_traits<A,B>::Type
```

```
binary_func(const A& a , const B& b){
    typedef CGAL::Coercion_traits<A,B> CT;

    // check for explicit interoperability
    CGAL_static_assertion((CT::Are_explicit_interoperable::value));

    // CT::Cast is used to to convert both types into the coercion type
    typename CT::Cast cast;
    // all operations are performed in the coercion type
    return cast(a)*cast(b);
}

int main(){
    // Function call for the interoperable types
    std::cout<< binary_func(double(3), int(5)) << std::endl;
    // Note that Coercion_traits is symmetric
    std::cout<< binary_func(int(3), double(5)) << std::endl;
    return 0;
}
```

File: examples/Algebraic_foundations/interoperable.cpp

The following example illustrates a dispatch for *ImplicitInteroperable* and *ExplicitInteroperable* types. The binary function (that just multiplies its two arguments) is supposed to take two *ExplicitInteroperable* arguments. For *ImplicitInteroperable* types a variant that avoids the explicit cast is selected.

```
#include <CGAL/basic.h>
#include <CGAL/Coercion_traits.h>
#include <CGAL/Quotient.h>
#include <CGAL/Sqrt_extension.h>
#include <CGAL/IO/io.h>

// this is the implementation for ExplicitInteroperable types
template <typename A, typename B>
typename CGAL::Coercion_traits<A,B>::Type
binary_function_(const A& a , const B& b, CGAL::Tag_false){
    std::cout << "Call for ExplicitInteroperable types: " << std::endl;
    typedef CGAL::Coercion_traits<A,B> CT;
    typename CT::Cast cast;
    return cast(a)*cast(b);
}

// this is the implementation for ImplicitInteroperable types
template <typename A, typename B>
typename CGAL::Coercion_traits<A,B>::Type
binary_function_(const A& a , const B& b, CGAL::Tag_true){
    std::cout << "Call for ImpicitInteroperable types: " << std::endl;
    return a*b;
}

// this function selects the correct implementation
template <typename A, typename B>
typename CGAL::Coercion_traits<A,B>::Type
```

```
binary_func(const A& a , const B& b){
    typedef CGAL::Coercion_traits<A,B> CT;
    typedef typename CT::Are_implicit_interoperable
   Are_implicit_interoperable;
    return binary_function_(a,b,Are_implicit_interoperable());
}

int main(){
    CGAL::set_pretty_mode(std::cout);

    // Function call for ImplicitInteroperable types
    std::cout<< binary_func(double(3), int(5)) << std::endl;

    // Function call for ExplicitInteroperable types
    CGAL::Quotient<int>           rational(1,3);    // == 1/3
    CGAL::Sqrt_extension<int,int> extension(1,2,3); // == 1+2*sqrt(3)
    CGAL::Sqrt_extension<CGAL::Quotient<int>,int> result =
   binary_func(rational, extension);
    std::cout<< result << std::endl;

    return 0;
}
```

File: examples/Algebraic_foundations/implicit_interoperable_dispatch.cpp

4.6 Fractions

Beyond the need for performing algebraic operations on objects as a whole, there are also number types which one would like to decompose into numerator and denominator. This does not only hold for rational numbers as *Quotient*, *Gmpq*, *mpq_class* or *leda_rational*, but also for compound objects as *Sqrt_extension* or *Polynomial* which may decompose into a (scalar) denominator and a compound numerator with a simpler coefficient type (e.g. integer instead of rational). Often operations can be performed faster on these denominator-free multiples. In case a type is a *Fraction* the relevant functionality as well as the numerator and denominator type are provided by *CGAL::Fraction_traits*. In particular *CGAL::Fraction_traits* provides a tag *Is_fraction* that can be used for dispatching.

A related class is *CGAL::Rational_traits* which has been kept for backward compatibility reasons. However, we recommend to use *Fraction_traits* since it is more general and offers dispatching functionality.

4.6.1 Examples

The following example show a simple use of *Fraction_traits*:

```
#include <CGAL/basic.h>
#include <CGAL/Fraction_traits.h>
#include <CGAL/IO/io.h>

#ifdef CGAL_USE_GMP
#include <CGAL/Gmpz.h>
```

```
#include <CGAL/Gmpq.h>
int main(){
    typedef CGAL::Fraction_traits<CGAL::Gmpq> FT;
    typedef FT::Numerator_type Numerator_type;
    typedef FT::Denominator_type Denominator_type;


   CGAL_static_assertion((boost::is_same<Numerator_type,CGAL::Gmpz>::value));

   CGAL_static_assertion((boost::is_same<Denominator_type,CGAL::Gmpz>::value));

    Numerator_type numerator;
    Denominator_type denominator;
    CGAL::Gmpq fraction(4,5);
    FT::Decompose()(fraction,numerator,denominator);

    CGAL::set_pretty_mode(std::cout);
    std::cout << "decompose fraction: "<< std::endl;
    std::cout << "fraction   : " << fraction << std::endl;
    std::cout << "numerator  : " << numerator<< std::endl;
    std::cout << "denominator: " << denominator << std::endl;

    std::cout << "re-compose fraction: "<< std::endl;
    fraction = FT::Compose()(numerator,denominator);
    std::cout << "fraction   : " << fraction << std::endl;
}
#else
int main(){ std::cout << "This examples needs GMP" << std::endl; }
#endif
```

File: examples/Algebraic_foundations/fraction_traits.cpp

The following example illustrates the integralization of a vector, i.e., the coefficient vector of a polynomial. Note that for minimizing coefficient growth *Fraction_traits<Type>::Common_factor* is used to compute the 'least' common multiple of the denominators.

```
#include <CGAL/basic.h>
#include <CGAL/Fraction_traits.h>
#include <CGAL/IO/io.h>
#include <vector>

template <class Fraction>
std::vector<typename CGAL::Fraction_traits<Fraction>::Numerator_type >
integralize(
        const std::vector<Fraction>& vec,
        typename CGAL::Fraction_traits<Fraction>::Denominator_type& d
) {
    typedef CGAL::Fraction_traits<Fraction> FT;
    typedef typename FT::Numerator_type Numerator_type;
    typedef typename FT::Denominator_type Denominator_type;
    typename FT::Decompose decompose;

    std::vector<Numerator_type>   num(vec.size());
```

```
    std::vector<Denominator_type> den(vec.size());

    // decompose each coefficient into integral part and denominator
    for (unsigned int i = 0; i < vec.size(); i++) {
        decompose(vec[i], num[i], den[i]);
    }

    // compute 'least' common multiple of all denominator
    // We would like to use gcd, so let's think of Common_factor as gcd.
    typename FT::Common_factor        gcd;
    d = 1;
    for (unsigned int i = 0; i < vec.size(); i++) {
        d *= CGAL::integral_division(den[i], gcd(d, den[i]));
    }

    // expand each (numerator, denominator) pair to common denominator
    for (unsigned int i = 0; i < vec.size(); i++) {
        // For simplicity ImplicitInteroperability is expected in this
   example
        num[i] *= CGAL::integral_division(d, den[i]);
    }
    return num;
}

#ifdef CGAL_USE_GMP

#include <CGAL/Gmpz.h>
#include <CGAL/Gmpq.h>

int main(){
    std::vector<CGAL::Gmpq> vec(3);
    vec[0]=CGAL::Gmpq(1,4);
    vec[1]=CGAL::Gmpq(1,6);
    vec[2]=CGAL::Gmpq(1,10);
    std::cout<< "compute an integralized vector" << std::endl;
    std::cout<<"input vector:  ["
             << vec[0] << "," << vec[1] << "," << vec[2] << "]" <<
   std::endl;
    CGAL::Gmpz d;
    std::vector<CGAL::Gmpz> integral_vec = integralize(vec,d);
    std::cout<<"output vector:  ["
             << integral_vec[0] << ","
             << integral_vec[1] << ","
             << integral_vec[2] << "]" << std::endl;
    std::cout<<"denominator  : "<< d <<std::endl;
}
#else
int main(){ std::cout << "This examples needs GMP" << std::endl; }
#endif
```

File: examples/Algebraic_foundations/integralize.cpp

4.7 Design and Implementation History

The package is part of CGAL since release 3.3. Of course the package is based on the former Number type support of CGAL. This goes back to Stefan Schirra and Andreas Fabri. But on the other hand the package is to a large extend influenced by the experience with the number type support in EXACUS [BEH+05], which in the main goes back to Lutz Kettner, Susan Hert, Arno Eigenwillig and Michael Hemmer. However, the package abstracts from the pure support for number types that are embedded on the real axis which allows the support of polynomials, finite fields, and algebraic extensions as well. See also related subsequent chapters.

Algebraic Foundations Reference Manual

Michael Hemmer

4.8 Classified Reference Pages

Algebraic Structures

Concepts

Classes

Global Functions

Real Embeddable

Concepts

Classes

Global Functions

Real Number Types

Concepts

Interoperability

Concepts

Classes

Fractions

Concepts

Classes

Miscellaneous

Concepts

4.9 Alphabetical List of Reference Pages

FromIntConstructible

Definition

A model of the concept *FromIntConstructible* is required to be constructible from int.

Creation

FromIntConstructible type(int& *i);*

Has Models

int
long
double

FromDoubleConstructible

Concept

Definition

A model of the concept *FromDoubleConstructible* is required to be constructible from the type *double*.

In case the type is a model of *RealEmbeddable* too, for any double d the identity: *d == CGAL::to_double(T(d))*, is guaranteed.

Creation

FromDoubleConstructible type(double d); conversion constructor from double.

Concept

ExplicitInteroperable

Definition

Two types *A* and *B* are a model of the *ExplicitInteroperable* concept, if it is possible to derive a superior type for *A* and *B*, such that both types are embeddable into this type. This type is *Coercion_traits<A,B>::Type*.

In this case *Coercion_traits<A,B>::Are_explicit_interoperable* is *Tag_true*.

A and *B* are valid argument types for all binary functors in *Algebraic_structure_traits<Type>* and *Real_embeddable_traits<Type>*. This is also the case for the respective global functions.

See Also

CGAL::Coercion_traits<A,B> . . . . page 68
ImplicitInteroperable . . . . page 67
AlgebraicStructureTraits . . . . page 78
RealEmbeddableTraits . . . . page 124

ImplicitInteroperable

Concept

Definition

Two types *A* and *B* are a model of the concept *ImplicitInteroperable*, if there is a superior type, such that binary arithmetic operations involving *A* and *B* result in this type. This type is *Coercion_traits<A,B>::Type*.

The type *Coercion_traits<A,B>::Type* is required to be implicit constructible from *A* and *B*.

In this case *Coercion_traits<A,B>::Are_implicit_interoperable* is *Tag_true*.

Refines

ExplicitInteroperable

See Also

CGAL::Coercion_traits<A,B> . . . . page 68
ExplicitInteroperable . . . . page 66
AlgebraicStructureTraits . . . . page 78
RealEmbeddableTraits . . . . page 124

Class

CGAL::Coercion_traits<A,B>

Definition

An instance of *Coercion_traits<A,B>* reflects the type coercion of the types A and B, it is symmetric in the two template arguments.

#include <CGAL/Coercion_traits.h>

Types

Coercion_traits<A,B>:: Are_explicit_interoperable

Tag indicating whether the two types A and B are a model of *ExplicitInteroperable*
This is either *CGAL::Tag_true* or *CGAL::Tag_false*.

Coercion_traits<A,B>:: Are_implicit_interoperable

Tag indicating whether the two types A and B are a model of *ImplicitInteroperable*
This is either *CGAL::Tag_true* or *CGAL::Tag_false*.

Coercion_traits<A,B>:: Type

The coercion type of *A* and *B*.
In case A and B are not *ExplicitInteroperable* this is undefined.

Coercion_traits<A,B>:: Cast

A model of the *AdaptableFunctor* concept, providing the conversion of *A* or *B* to *Type*.
In case A and B are not *ExplicitInteroperable* this is undefined.

See Also

ExplicitInteroperable . . . . page 66
ImplicitInteroperable . . . . page 67

IntegralDomainWithoutDivision

Definition

This is the most basic concept for algebraic structures considered within CGAL.

A model *IntegralDomainWithoutDivision* represents an integral domain, i.e. commutative ring with 0, 1, +, * and unity free of zero divisors.
Note: A model is not required to offer the always well defined integral division.

It refines *Assignable*, *CopyConstructible*, *DefaultConstructible* and *FromIntConstructible*.
It refines *EqualityComparable*, where equality is defined w.r.t. the ring element being represented.
The operators unary and binary plus +, unary and binary minus -, multiplication * and their compound forms +=, -=, *= are required and implement the respective ring operations.

Moreover, *CGAL::Algebraic_structure_traits< IntegralDomainWithoutDivision >* is a model of *AlgebraicStructureTraits* providing:
- *CGAL::Algebraic_structure_traits< IntegralDomainWithoutDivision >::Algebraic_type* derived from *Integral_domain_without_division_tag*
- *CGAL::Algebraic_structure_traits< IntegralDomainWithoutDivision >::Is_zero*
- *CGAL::Algebraic_structure_traits< IntegralDomainWithoutDivision >::Is_one*
- *CGAL::Algebraic_structure_traits< IntegralDomainWithoutDivision >::Square*
- *CGAL::Algebraic_structure_traits< IntegralDomainWithoutDivision >::Simplify*
- *CGAL::Algebraic_structure_traits< IntegralDomainWithoutDivision >::Unit_part*

Refines

Assignable
CopyConstructible
DefaultConstructible
EqualityComparable

FromIntConstructible

Operations

IntegralDomainWithoutDivision

$+a$ — unary plus

IntegralDomainWithoutDivision

$-a$ — unary minus

IntegralDomainWithoutDivision

$a+b$

IntegralDomainWithoutDivision

$a-b$

IntegralDomainWithoutDivision	$a*b$	
IntegralDomainWithoutDivision	$a += b$	
IntegralDomainWithoutDivision	$a -= b$	
IntegralDomainWithoutDivision	$a *= b$	

Equality comparable:

result_type	$a == b$	The *result_type* is convertible to *bool*.
result_type	$a\ != b$	The *result_type* is convertible to *bool*.

See Also

IntegralDomain

Concept

Definition

IntegralDomain refines *IntegralDomainWithoutDivision* by providing an integral division.
Note: The concept does not require the operator / for this operation. We intend to reserve the operator syntax for use with a *Field*.

Moreover, *CGAL::Algebraic_structure_traits*< *IntegralDomain* > is a model of *AlgebraicStructureTraits* providing:

- *CGAL::Algebraic_structure_traits*< *IntegralDomain* >*::Algebraic_type* derived from *Integral_domain_tag*
- *CGAL::Algebraic_structure_traits*< *IntegralDomain* >*::Integral_division*
- *CGAL::Algebraic_structure_traits*< *IntegralDomain* >*::Divides*

Refines

IntegralDomainWithoutDivision

See Also

Concept

UniqueFactorizationDomain

Definition

A model of *UniqueFactorizationDomain* is an *IntegralDomain* with the additional property that the ring it represents is a unique factorization domain (a.k.a. UFD or factorial ring), meaning that every non-zero non-unit element has a factorization into irreducible elements that is unique up to order and up to multiplication by invertible elements (units). (An irreducible element is a non-unit ring element that cannot be factored further into two non-unit elements. In a UFD, the irreducible elements are precisely the prime elements.)

In a UFD, any two elements, not both zero, possess a greatest common divisor (gcd).

Moreover, *CGAL::Algebraic_structure_traits*< *UniqueFactorizationDomain* > is a model of *AlgebraicStructureTraits* providing:
- *CGAL::Algebraic_structure_traits*< *UniqueFactorizationDomain* >*::Algebraic_type* derived from *Unique_factorization_domain_tag*
- *CGAL::Algebraic_structure_traits*< *UniqueFactorizationDomain* >*::Gcd*

Refines

IntegralDomain

See Also

EuclideanRing

Concept

Definition

A model of *EuclideanRing* represents an euclidean ring (or Euclidean domain). It is an *UniqueFactorizationDomain* that affords a suitable notion of minimality of remainders such that given x and $y \neq 0$ we obtain an (almost) unique solution to $x = qy + r$ by demanding that a solution (q, r) is chosen to minimize r. In particular, r is chosen to be 0 if possible.

Moreover, *CGAL::Algebraic_structure_traits< EuclideanRing >* is a model of *AlgebraicStructureTraits* providing:

- *CGAL::Algebraic_structure_traits< EuclideanRing >::Algebraic_type* derived from *Unique_factorization_domain_tag*
- *CGAL::Algebraic_structure_traits< EuclideanRing >::Mod*
- *CGAL::Algebraic_structure_traits< EuclideanRing >::Div*
- *CGAL::Algebraic_structure_traits< EuclideanRing >::Div_mod*

Remarks

The most prominent example of a Euclidean ring are the integers. Whenever both x and y are positive, then it is conventional to choose the smallest positive remainder r.

Refines

UniqueFactorizationDomain

See Also

IntegralDomainWithoutDivision . . . . . . page 69
IntegralDomain . . . . . . page 71
UniqueFactorizationDomain . . . . . . page 72
EuclideanRing . . . . . . page 73
Field . . . . . . page 74
FieldWithSqrt . . . . . . page 75
FieldWithKthRoot . . . . . . page 76
FieldWithRootOf . . . . . . page 77
AlgebraicStructureTraits . . . . . . page 78

Concept

Field

Definition

A model of *Field* is an *IntegralDomain* in which every non-zero element has a multiplicative inverse. Thus, one can divide by any non-zero element. Hence division is defined for any divisor != 0. For a Field, we require this division operation to be available through operators / and /=.

Moreover, *CGAL::Algebraic_structure_traits*< *Field* > is a model of *AlgebraicStructureTraits* providing:
- *CGAL::Algebraic_structure_traits*< *Field* >*::Algebraic_type* derived from *Field_tag*
- *CGAL::Algebraic_structure_traits*< *FieldWithSqrt* >*::Inverse*

Refines

IntegralDomain

Operations

Field	a/b
Field	$a/=b$

See Also

FieldWithSqrt

Concept

Definition

A model of *FieldWithSqrt* is a *Field* that has operations to take square roots.

Moreover, *CGAL::Algebraic_structure_traits*< *FieldWithSqrt* > is a model of *AlgebraicStructureTraits* providing:
- *CGAL::Algebraic_structure_traits*< *FieldWithSqrt* >*::Algebraic_type* derived from *Field_with_sqrt_tag*
- *CGAL::Algebraic_structure_traits*< *FieldWithSqrt* >*::Sqrt*

Refines

Field

See Also

Concept

FieldWithKthRoot

Definition

A model of *FieldWithKthRoot* is a *FieldWithSqrt* that has operations to take k-th roots.

Moreover, *CGAL::Algebraic_structure_traits< FieldWithKthRoot >* is a model of *AlgebraicStructureTraits* providing:
- *CGAL::Algebraic_structure_traits< FieldWithKthRoot >::Algebraic_type* derived from *Field_with_kth_root_tag*
- *CGAL::Algebraic_structure_traits< FieldWithKthRoot >::Kth_root*

Refines

FieldWithSqrt

See Also

IntegralDomainWithoutDivision . . . . . page 69
IntegralDomain . . . . . page 71
UniqueFactorizationDomain . . . . . page 72
EuclideanRing . . . . . page 73
Field . . . . . page 74
FieldWithSqrt . . . . . page 75
FieldWithKthRoot . . . . . page 76
FieldWithRootOf . . . . . page 77
AlgebraicStructureTraits . . . . . page 78

FieldWithRootOf

Concept

Definition

A model of *FieldWithRootOf* is a *FieldWithKthRoot* with the possibility to construct it as the root of a univariate polynomial.

Moreover, *CGAL::Algebraic_structure_traits*< *FieldWithRootOf* > is a model of *AlgebraicStructureTraits* providing:
- *CGAL::Algebraic_structure_traits*< *FieldWithRootOf* >*::Algebraic_type* derived from *Field_with_kth_root_tag*
- *CGAL::Algebraic_structure_traits*< *FieldWithRootOf* >*::Root_of*

Refines

FieldWithKthRoot

See Also

IntegralDomainWithoutDivision . . . page 69
IntegralDomain . . . page 71
UniqueFactorizationDomain . . . page 72
EuclideanRing . . . page 73
Field . . . page 74
FieldWithSqrt . . . page 75
FieldWithKthRoot . . . page 76
FieldWithRootOf . . . page 77
AlgebraicStructureTraits . . . page 78

Concept

AlgebraicStructureTraits

Definition

A model of *AlgebraicStructureTraits* reflects the algebraic structure of an associated type *Type*.

Depending on the concepts that *Type* fulfills, it contains various functors and descriptive tags. Moreover it gives access to the several possible algebraic operations within that structure.

Types

A model of *AlgebraicStructureTraits* is supposed to provide:

AlgebraicStructureTraits:: Type

The associated type.

AlgebraicStructureTraits:: Algebraic_category

Tag indicating the algebraic structure of the associated type.

Tag is:	*Type* is model of:
CGAL::Null_tag	no algebraic concept
CGAL::Integral_domain_without_division_tag	*IntegralDomainWithoutDivision*
CGAL::Integral_domain_tag	*IntegralDomain*
CGAL::Unique_factorization_domain_tag	*UniqueFactorizationDomain*
CGAL::Euclidean_ring_tag	*EuclideanRing*
CGAL::Field_tag	*Field*
CGAL::Field_with_sqrt_tag	*FieldWithSqrt*
CGAL::Field_with_kth_root_tag	*FieldWithKthRoot*
CGAL::Field_with_root_of_tag	*FieldWithRootOf*

AlgebraicStructureTraits:: Is_exact

Tag indicating whether *Type* is exact.
This is either *CGAL::Tag_true* or *CGAL::Tag_false*.
An algebraic structure is considered exact, if all operations required by its concept are computed such that a comparison of two algebraic expressions is always correct. The exactness covers only those operations that are required by the algebraic structure concept.
e.g. an exact *Field* may have a *Sqrt* functor that is not exact.

AlgebraicStructureTraits:: Is_numerical_sensitive

Tag indicating whether *Type* is numerical sensitive.
This is either *CGAL::Tag_true* or *CGAL::Tag_false*.
An algebraic structure is considered as numerically sensitive, if the performance of the type is sensitive to the condition number of an algorithm.

AlgebraicStructureTraits:: Boolean

This type specifies the return type of the predicates provided by this traits. The type must be convertible to *bool* and typically the type indeed maps to *bool*. However, there are also cases such as interval arithmetic, in which it is *Uncertain<bool>* or some similar type.

Functors

In case a functor is not provided, it is set to *CGAL::Null_functor*.

AlgebraicStructureTraits:: Is_zero

A model of *AlgebraicStructureTraits::IsZero*.
Required by the concept *IntegralDomainWithoutDivision*. In case *Type* is also model of *RealEmbeddable* this is a model of *RealEmbeddableTraits::IsZero*.

AlgebraicStructureTraits:: Is_one

A model of *AlgebraicStructureTraits::IsOne*.
Required by the concept *IntegralDomainWithoutDivision*.

AlgebraicStructureTraits:: Square

A model of *AlgebraicStructureTraits::Square*.
Required by the concept *IntegralDomainWithoutDivision*.

AlgebraicStructureTraits:: Simplify

A model of *AlgebraicStructureTraits::Simplify*.
Required by the concept *IntegralDomainWithoutDivision*.

AlgebraicStructureTraits:: Unit_part

A model of *AlgebraicStructureTraits::UnitPart*.
Required by the concept *IntegralDomainWithoutDivision*.

AlgebraicStructureTraits:: Integral_division

A model of *AlgebraicStructureTraits::IntegralDivision*.
Required by the concept *IntegralDomain*.

AlgebraicStructureTraits:: Divides

A model of *AlgebraicStructureTraits::Divides*.
Required by the concept *IntegralDomain*.

AlgebraicStructureTraits:: Is_square

A model of *AlgebraicStructureTraits::IsSquare*.
Required by the concept *IntegralDomainWithoutDivision*.

AlgebraicStructureTraits:: Gcd

A model of *AlgebraicStructureTraits::Gcd*.
Required by the concept *UniqueFactorizationDomain*.

AlgebraicStructureTraits:: Mod

A model of *AlgebraicStructureTraits::Mod*.
Required by the concept *EuclideanRing*.

AlgebraicStructureTraits:: Div

A model of *AlgebraicStructureTraits::Div*.
Required by the concept *EuclideanRing*.

AlgebraicStructureTraits:: Div_mod

A model of *AlgebraicStructureTraits::DivMod*.
Required by the concept *EuclideanRing*.

AlgebraicStructureTraits:: Inverse

A model of *AlgebraicStructureTraits::Inverse*.
Required by the concept *Field*.

AlgebraicStructureTraits:: Sqrt

A model of *AlgebraicStructureTraits::Sqrt*.
Required by the concept *FieldWithSqrt*.

AlgebraicStructureTraits:: Kth_root

A model of *AlgebraicStructureTraits::KthRoot*.
Required by the concept *FieldWithKthRoot*.

AlgebraicStructureTraits:: Root_of

A model of *AlgebraicStructureTraits::RootOf*.
Required by the concept *FieldWithRootOf*.

See Also

IntegralDomainWithoutDivision . . . . page 69
IntegralDomain . . . . page 71
UniqueFactorizationDomain . . . . page 72
EuclideanRing . . . . page 73
Field . . . . page 74

Has Models

AlgebraicStructureTraits::IsZero

Definition

AdaptableUnaryFunction, returns true in case the argument is the zero element of the ring.

Refines

AdaptableUnaryFunction

Types

AlgebraicStructureTraits::IsZero:: result_type Is *AlgebraicStructureTraits::Boolean*.

AlgebraicStructureTraits::IsZero:: argument_type

Is *AlgebraicStructureTraits::Type*.

Operations

result_type *is_zero(argument_type x)*

returns true in case *x* is the zero element of the ring.

See Also

AlgebraicStructureTraits .... page 78 *RealEmbeddableTraits::IsZero* .... page 126

AlgebraicStructureTraits::IsOne

Definition

AdaptableUnaryFunction, returns true in case the argument is the one of the ring.

Refines

AdaptableUnaryFunction

Types

AlgebraicStructureTraits::IsOne:: result_type Is *AlgebraicStructureTraits::Boolean*.

AlgebraicStructureTraits::IsOne:: argument_type

Is *AlgebraicStructureTraits::Type*.

Operations

result_type *is_one(argument_type x)*

returns true in case x is the one of the ring.

See Also

AlgebraicStructureTraits . . . . page 78

AlgebraicStructureTraits::Square

Definition

AdaptableUnaryFunction, computing the square of the argument.

Refines

AdaptableUnaryFunction

Types

AlgebraicStructureTraits::Square:: result_type

Is *AlgebraicStructureTraits::Type*.

AlgebraicStructureTraits::Square:: argument_type

Is *AlgebraicStructureTraits::Type*.

Operations

result_type *square(argument_type x)*

returns the square of x.

See Also

AlgebraicStructureTraits . . . . page 78

AlgebraicStructureTraits::Simplify

Functor
Concept

Definition

This *AdaptableUnaryFunction* may simplify a given object.

Refines

AdaptableUnaryFunction

Types

AlgebraicStructureTraits::Simplify:: result_type

Is void.

AlgebraicStructureTraits::Simplify:: argument_type

Is *AlgebraicStructureTraits::Type*.

Operations

result_type *simplify(argument_type x)*

may simplify x.

See Also

AlgebraicStructureTraits . . . . page 78

AlgebraicStructureTraits::UnitPart

Definition

This *AdaptableUnaryFunction* computes the unit part of a given ring element.

The mathematical definition of unit part is as follows: Two ring elements a and b are said to be associate if there exists an invertible ring element (i.e. a unit) u such that $a = ub$. This defines an equivalence relation. We can distinguish exactly one element of every equivalence class as being unit normal. Then each element of a ring possesses a factorization into a unit (called its unit part) and a unit-normal ring element (called its unit normal associate).

For the integers, the non-negative numbers are by convention unit normal, hence the unit-part of a non-zero integer is its sign. For a *Field*, every non-zero element is a unit and is its own unit part, its unit normal associate being one. The unit part of zero is, by convention, one.

Refines

AdaptableUnaryFunction

Types

AlgebraicStructureTraits::UnitPart:: result_type

Is *AlgebraicStructureTraits::Type*.

AlgebraicStructureTraits::UnitPart:: argument_type

Is *AlgebraicStructureTraits::Type*.

Operations

result_type *unit_part(argument_type x)*

returns the unit part of x.

See Also

AlgebraicStructureTraits . . . . page 78

AlgebraicStructureTraits::IntegralDivision

Definition

AdaptableBinaryFunction providing an integral division.

Integral division (a.k.a. exact division or division without remainder) maps ring elements (x,y) to ring element z such that $x = yz$ if such a z exists (i.e. if x is divisible by y). Otherwise the effect of invoking this operation is undefined. Since the ring represented is an integral domain, z is uniquely defined if it exists.

Refines

AdaptableBinaryFunction

Types

AlgebraicStructureTraits::IntegralDivision:: result_type

Is *AlgebraicStructureTraits::Type*.

AlgebraicStructureTraits::IntegralDivision:: first_argument

Is *AlgebraicStructureTraits::Type*.

AlgebraicStructureTraits::IntegralDivision:: second_argument

Is *AlgebraicStructureTraits::Type*.

Operations

result_type   *integral_division(first_argument_type x, second_argument_type y)*

returns x/y, this is an integral division.

template <class NT1, class NT2>
result_type   *integral_division(NT1 x, NT2 y)*

This operator is defined if *NT1* and *NT2* are *ExplicitInteroperable* with coercion type *AlgebraicStructureTraits::Type*.

See Also

AlgebraicStructureTraits . . . . page 78
AlgebraicStructureTraits::Divides . . . . page 88

Functor
Concept

AlgebraicStructureTraits::Divides

Definition

AdaptableBinaryFunction, returns true if the first argument divides the second argument.

Integral division (a.k.a. exact division or division without remainder) maps ring elements (n,d) to ring element c such that $n = dc$ if such a c exists. In this case it is said that d divides n.

This functor is required to provide two operators. The first operator takes two arguments and returns true if the first argument divides the second argument. The second operator returns c via the additional third argument.

Refines

AdaptableBinaryFunction

Types

AlgebraicStructureTraits::Divides:: result_type

Is *AlgebraicStructureTraits::Boolean*.

AlgebraicStructureTraits::Divides:: first_argument

Is *AlgebraicStructureTraits::Type*.

AlgebraicStructureTraits::Divides:: second_argument

Is *AlgebraicStructureTraits::Type*.

Operations

result_type *divides(first_argument_type d, second_argument_type n)*

Computes whether d divides n.

result_type *divides(first_argument_type d, second_argument_type n, AlgebraicStructureTraits::Type& c)*

Computes whether d divides n. Moreover it computes c if d divides n, otherwise the value of c is undefined.

See Also

AlgebraicStructureTraits .... page 78
AlgebraicStructureTraits::IntegralDivision .... page 87

AlgebraicStructureTraits::IsSquare

Definition

AdaptableBinaryFunction that computes whether the first argument is a square. If the first argument is a square the second argument, which is taken by reference, contains the square root. Otherwise, the content of the second argument is undefined.

A ring element x is said to be a square iff there exists a ring element y such that $x = y * y$. In case the ring is a *UniqueFactorizationDomain*, y is uniquely defined up to multiplication by units.

Refines

AdaptableBinaryFunction

Types

AlgebraicStructureTraits::IsSquare:: result_type

Is *AlgebraicStructureTraits::Boolean*.

AlgebraicStructureTraits::IsSquare:: first_argument

Is *AlgebraicStructureTraits::Type*.

AlgebraicStructureTraits::IsSquare:: second_argument

Is *AlgebraicStructureTraits::Type*`&`.

Operations

result_type *is_square(first_argument_type x, second_argument_type y)*

returns `true` in case x is a square, i.e. $x = y * y$.
Postcondition: $unit\_part(y) == 1$.

result_type *is_square(first_argument_type x)*

returns `true` in case x is a square.

See Also

AlgebraicStructureTraits . . . . page 78

AlgebraicStructureTraits::Gcd

Definition

AdaptableBinaryFunction providing the gcd.

The greatest common divisor (*gcd*) of ring elements x and y is the unique ring element d (up to a unit) with the property that any common divisor of x and y also divides d. (In other words: d is the greatest lower bound of x and y in the partial order of divisibility.) We demand the *gcd* to be unit-normal (i.e. have unit part 1).

$gcd(0,0)$ is defined as 0, since 0 is the greatest element with respect to the partial order of divisibility. This is because an element $a \in R$ is said to divide $b \in R$, iff $\exists r \in R$ such that $a \cdot r = b$. Thus, 0 is divided by every element of the Ring, in particular by itself.

Refines

AdaptableBinaryFunction

Types

AlgebraicStructureTraits::Gcd:: result_type — Is *AlgebraicStructureTraits::Type*.

AlgebraicStructureTraits::Gcd:: first_argument — Is *AlgebraicStructureTraits::Type*.

AlgebraicStructureTraits::Gcd:: second_argument — Is *AlgebraicStructureTraits::Type*.

Operations

result_type gcd(first_argument_type x, second_argument_type y)

returns $gcd(x,y)$.

template <class NT1, class NT2>
result_type gcd(NT1 x, NT2 y)

This operator is defined if *NT1* and *NT2* are *ExplicitInteroperable* with coercion type *AlgebraicStructureTraits::Type*.

See Also

AlgebraicStructureTraits . . . . page 78

AlgebraicStructureTraits::DivMod

Definition

AdaptableFunctor computes both integral quotient and remainder of division with remainder. The quotient q and remainder r are computed such that $x = q * y + r$ and $|r| < |y|$ with respect to the proper integer norm of the represented ring. [1] In particular, r is chosen to be 0 if possible. Moreover, we require q to be minimized with respect to the proper integer norm.

Note that the last condition is needed to ensure a unique computation of the pair (q, r). However, an other option is to require minimality for $|r|$, with the advantage that a *mod(x,y)* operation would return the unique representative of the residue class of x with respect to y, e.g. $mod(2,3)$ should return -1. But this conflicts with nearly all current implementation of integer types. From there, we decided to stay conform with common implementations and require q to be computed as x/y rounded towards zero.

The following table illustrates the behavior for integers:

x	y	q	r
3	3	1	0
2	3	0	2
1	3	0	1
0	3	0	0
-1	3	0	-1
-2	3	0	-2
-3	3	-1	0

-

x	y	q	r
3	-3	-1	0
2	-3	0	2
1	-3	0	1
0	-3	0	0
-1	-3	0	-1
-2	-3	0	-2
-3	-3	1	0

Refines

AdaptableFunctor

Types

AlgebraicStructureTraits::DivMod:: result_type

Is void.

AlgebraicStructureTraits::DivMod:: first_argument_type

Is *AlgebraicStructureTraits::Type*.

AlgebraicStructureTraits::DivMod:: second_argument_type

Is *AlgebraicStructureTraits::Type*.

AlgebraicStructureTraits::DivMod:: third_argument_type

Is *AlgebraicStructureTraits::Type*&.

AlgebraicStructureTraits::DivMod:: fourth_argument_type

Is *AlgebraicStructureTraits::Type*&.

[1] For integers this norm is the absolute value.
For univariate polynomials this norm is the degree.

Operations

result_type *div_mod.operator()(first_argument_type x,*
second_argument_type y,
third_argument_type q,
fourth_argument_type r)

computes the quotient q and remainder r, such that $x = q * y + r$ and r minimal with respect to the Euclidean Norm on *Type*.

template <class NT1, class NT2>
result_type *div_mod(NT1 x, NT2 y, third_argument_type q, fourth_argument_type r)*

This operator is defined if *NT1* and *NT2* are *ExplicitInteroperable* with coercion type *AlgebraicStructureTraits::Type*.

See Also

AlgebraicStructureTraits . . . . page 78
AlgebraicStructureTraits::Mod . . . . page 94
AlgebraicStructureTraits::Div . . . . page 93

AlgebraicStructureTraits::Div

Definition

AdaptableBinaryFunction computes the integral quotient of division with remainder.

Refines

AdaptableBinaryFunction

Types

AlgebraicStructureTraits::Div:: result_type	Is *AlgebraicStructureTraits::Type*.
AlgebraicStructureTraits::Div:: first_argument	Is *AlgebraicStructureTraits::Type*.
AlgebraicStructureTraits::Div:: second_argument	Is *AlgebraicStructureTraits::Type*.

Operations

result_type *div(first_argument_type x, second_argument_type y)*

template <class NT1, class NT2>
result_type *div(NT1 x, NT2 y)*

This operator is defined if *NT1* and *NT2* are *ExplicitInteroperable* with coercion type *AlgebraicStructureTraits::Type*.

See Also

AlgebraicStructureTraits . . . . page 78
AlgebraicStructureTraits::Mod . . . . page 94
AlgebraicStructureTraits::DivMod . . . . page 91

AlgebraicStructureTraits::Mod

Definition

AdaptableBinaryFunction computes the remainder of division with remainder.

Refines

AdaptableBinaryFunction

Types

AlgebraicStructureTraits::Mod:: result_type Is *AlgebraicStructureTraits::Type*.

AlgebraicStructureTraits::Mod:: first_argument

Is *AlgebraicStructureTraits::Type*.

AlgebraicStructureTraits::Mod:: second_argument

Is *AlgebraicStructureTraits::Type*.

Operations

result_type *mod(first_argument_type x, second_argument_type y)*

template <class NT1, class NT2>
result_type *mod(NT1 x, NT2 y)*

This operator is defined if *NT1* and *NT2* are *ExplicitInteroperable* with coercion type *AlgebraicStructureTraits::Type*.

See Also

AlgebraicStructureTraits . . . . page 78
AlgebraicStructureTraits::Div . . . . page 93
AlgebraicStructureTraits::DivMod . . . . page 91

AlgebraicStructureTraits::Inverse

Definition

AdaptableUnaryFunction providing the inverse element with respect to multiplication of a *Field*.

Refines

AdaptableUnaryFunction

Types

AlgebraicStructureTraits::Inverse:: result_type

Is *AlgebraicStructureTraits::Type*.

AlgebraicStructureTraits::Inverse:: argument_type

Is *AlgebraicStructureTraits::Type*.

Operations

result_type *inverse(argument_type x)*

returns the inverse element of x with respect to multiplication.
Precondition: $x \neq 0$

See Also

AlgebraicStructureTraits . . . . page 78

Functor Concept

AlgebraicStructureTraits::Sqrt

Definition

AdaptableUnaryFunction providing the square root.

Refines

AdaptableUnaryFunction

Types

AlgebraicStructureTraits::Sqrt:: result_type Is *AlgebraicStructureTraits::Type*.

AlgebraicStructureTraits::Sqrt:: argument_type

Is *AlgebraicStructureTraits::Type*.

Operations

result_type *sqrt(argument_type x)*

returns $\sqrt{x}$.

See Also

AlgebraicStructureTraits . . . . page 78

AlgebraicStructureTraits::KthRoot

Functor
Concept

Definition

AdaptableBinaryFunction providing the k-th root.

Refines

AdaptableBinaryFunction

Types

AlgebraicStructureTraits::KthRoot:: result_type

Is *AlgebraicStructureTraits::Type*.

AlgebraicStructureTraits::KthRoot:: first_argument

Is int.

AlgebraicStructureTraits::KthRoot:: second_argument

Is *AlgebraicStructureTraits::Type*.

Operations

result_type *kth_root(int k, second_argument_type x)*

returns the k-th root of x.
Precondition: $k \geq 1$

See Also

FieldWithRootOf . . . . page 77
AlgebraicStructureTraits . . . . page 78

AlgebraicStructureTraits::RootOf

Definition

AdaptableFunctor computes a real root of a square-free univariate polynomial.

Refines

AdaptableFunctor

Types

AlgebraicStructureTraits::RootOf :: result_type

Is *AlgebraicStructureTraits::Type*.

Operations

template<class InputIterator>
result_type *root_of(int k, InputIterator begin, InputIterator end)*

returns the k-th real root of the univariate polynomial, which is defined by the iterator range, where begin refers to the constant term.
Precondition: The polynomial is square-free.
Precondition: The value type of the InputIterator is *AlgebraicStructureTraits::Type*

See Also

FieldWithRootOf . . . . page 77
AlgebraicStructureTraits . . . . page 78

CGAL::Algebraic_structure_traits<T>

Class

Definition

An instance of *Algebraic_structure_traits<T>* is a model of *AlgebraicStructureTraits*, where T is the associated type.

#include <CGAL/Algebraic_structure_traits.h>

Is Model for the Concepts

AlgebraicStructureTraits . . . . page 78

Class

CGAL::Integral_domain_without_division_tag

Definition

Tag indicating that a type is a model of the *IntegralDomainWithoutDivision* concept.

Is Model for the Concepts

DefaultConstructible

See Also

IntegralDomainWithoutDivision . . . . page 69

CGAL::Integral_domain_tag

Class

Definition

Tag indicating that a type is a model of the *IntegralDomain* concept.

Inherits From

Integral_domain_without_division_tag

Is Model for the Concepts

DefaultConstructible

See Also

IntegralDomain . . . . page 71
AlgebraicStructureTraits . . . . page 78

Class

CGAL::Unique_factorization_domain_tag

Definition

Tag indicating that a type is a model of the *UniqueFactorizationDomain* concept.

Inherits From

Integral_domain_tag

Is Model for the Concepts

DefaultConstructible

See Also

UniqueFactorizationDomain . . . . page 72
AlgebraicStructureTraits . . . . page 78

CGAL::Euclidean_ring_tag

Class

Definition

Tag indicating that a type is a model of the *EuclideanRing* concept.

Inherits From

Unique_factorization_domain_tag

Is Model for the Concepts

DefaultConstructible

See Also

EuclideanRing . . . . page 73
AlgebraicStructureTraits . . . . page 78

CGAL::Field_tag

Definition

Tag indicating that a type is a model of the *Field* concept.

Inherits From

Integral_domain_tag

Is Model for the Concepts

DefaultConstructible

See Also

Field . . . . page 74
AlgebraicStructureTraits . . . . page 78

CGAL::Field_with_sqrt_tag

Class

Definition

Tag indicating that a type is a model of the *FieldWithSqrt* concept.

Inherits From

Field_tag

Is Model for the Concepts

DefaultConstructible

See Also

FieldWithSqrt . . . . page 75
AlgebraicStructureTraits . . . . page 78

Class

CGAL::Field_with_kth_root_tag

Definition

Tag indicating that a type is a model of the *FieldWithKthRoot* concept.

Inherits From

Field_with_sqrt_tag

Is Model for the Concepts

DefaultConstructible

See Also

FieldWithKthRoot . . . . page 76
AlgebraicStructureTraits . . . . page 78

CGAL::Field_with_root_of_tag

Class

Definition

Tag indicating that a type is a model of the *FieldWithRootOf* concept.

Inherits From

Field_with_kth_root_tag

Is Model for the Concepts

DefaultConstructible

See Also

FieldWithRootOf . . . . page 77
AlgebraicStructureTraits . . . . page 78

CGAL::is_zero

Definition

The function *is_zero* determines if a value is equal to 0 or not.
The function is defined if the argument type is a model of the *RealEmbeddable* or of the *IntegralDomainWithoutDivision* concept.

#include <CGAL/number_utils.h>

template <class NT>

result_type	*is_zero(NT x)*	The *result_type* is convertible to *bool*.

See Also

RealEmbeddable . . . . page 123
RealEmbeddableTraits::IsZero . . . . page 126
IntegralDomainWithoutDivision . . . . page 69
AlgebraicStructureTraits::IsZero . . . . page 82

CGAL::is_one

Function

Definition

The function *is_one* determines if a value is equal to 1 or not.
The function is defined if the argument type is a model of the *IntegralDomainWithoutDivision* concept.

#include <CGAL/number_utils.h>

template <class NT>		
result_type	*is_one(NT x)*	The *result_type* is convertible to *bool*.

See Also

IntegralDomainWithoutDivision . . . . page 69
AlgebraicStructureTraits::IsOne . . . . page 83

CGAL::square

Definition

The function *square* returns the square of a number.
The function is defined if the argument type is a model of the *IntegralDomainWithoutDivision* concept.

#include <CGAL/number_utils.h>

template <class NT>
NT *square(NT x)*

See Also

IntegralDomainWithoutDivision . . . . page 69
AlgebraicStructureTraits::Square . . . . page 84

CGAL::is_square

Function

Definition

An ring element x is said to be a square iff there exists a ring element y such that $x = y * y$. In case the ring is a *UniqueFactorizationDomain*, y is uniquely defined up to multiplication by units.

The function *is_square* is available if *Algebraic_structure_traits::Is_square* is not the *CGAL::Null_functor*.

#include <CGAL/number_utils.h>

template <class NT>
result_type *is_square(NT x)* The *result_type* is convertible to *bool*.

template <class NT>
result_type *is_square(NT x, NT& y)*

The *result_type* is convertible to *bool*.

See Also

UniqueFactorizationDomain . . . . page 72
AlgebraicStructureTraits::IsSquare . . . . page 89

Function

CGAL::simplify

Definition

The function *simplify* may simplify a given object.

The function is defined if the argument type is a model of the *IntegralDomainWithoutDivision* concept.

#include <CGAL/number_utils.h>

template <class NT>
void *simplify(NT x)*

See Also

IntegralDomainWithoutDivision . . . . page 69
AlgebraicStructureTraits::Simplify . . . . page 85

CGAL::unit_part

Definition

The function *unit_part* computes the unit part of a given ring element.

The function is defined if the argument type is a model of the *IntegralDomainWithoutDivision* concept.

#include <CGAL/number_utils.h>

template <class NT>
NT *unit_part(NT x)*

See Also

IntegralDomainWithoutDivision . . . . page 69
AlgebraicStructureTraits::UnitPart . . . . page 86

Function

CGAL::integral_division

Definition

The function *integral_division* (a.k.a. exact division or division without remainder) maps ring elements (x, y) to ring element z such that $x = yz$ if such a z exists (i.e. if x is divisible by y). Otherwise the effect of invoking this operation is undefined. Since the ring represented is an integral domain, z is uniquely defined if it exists.

In case the argument types *NT1* and *NT2* differ, the *result_type* is determined via *Coercion_traits*.
Thus, the *result_type* is well defined if *NT1* and *NT2* are a model of *ExplicitInteroperable*.
The actual *integral_division* is performed with the semantic of that type.

The function is defined if *result_type* is a model of the *IntegralDomain* concept.

#include <CGAL/number_utils.h>

template <class NT1, class NT2>
result_type *integral_division(NT1 x, NT2 y)*

See Also

IntegralDomain . . . . page 71
AlgebraicStructureTraits::IntegralDivision . . . . page 87

CGAL::gcd

Function

Definition

The function *gcd* computes the greatest common divisor of two values.

In case the argument types *NT1* and *NT2* differ, the *result_type* is determined via *Coercion_traits*.
Thus, the *result_type* is well defined if *NT1* and *NT2* are a model of *ExplicitInteroperable*.
The actual *gcd* is performed with the semantic of that type.

The function is defined if *result_type* is a model of the *UniqueFactorizationDomain* concept.

#include <CGAL/number_utils.h>

template <class NT1, class NT2>
result_type *gcd(NT1 x, NT2 y)*

See Also

UniqueFactorizationDomain . . . . page 72
AlgebraicStructureTraits::Gcd . . . . page 90

Function

CGAL::div_mod

Definition

The function *div_mod* computes the integral quotient and remainder of division with remainder.

In case the argument types *NT1* and *NT2* differ, the *result_type* is determined via *Coercion_traits*.
Thus, the *result_type* is well defined if *NT1* and *NT2* are a model of *ExplicitInteroperable*.
The actual *div_mod* is performed with the semantic of that type.

The function is defined if *result_type* is a model of the *EuclideanRing* concept.

#include <CGAL/number_utils.h>

template <class NT1, class NT2>
void *div_mod(NT1 x, NT2 y, result_type& q, result_type& r)*

computes the quotient q and remainder r, such that $x = q * y + r$ and r minimal with respect to the Euclidean Norm of the *result_type*.

See Also

EuclideanRing . . . . page 73
AlgebraicStructureTraits::DivMod . . . . page 91
CGAL::mod . . . . page 118
CGAL::div . . . . page 117

CGAL::div

Function

Definition

The function *div* computes the integral quotient of division with remainder.

In case the argument types *NT1* and *NT2* differ, the *result_type* is determined via *Coercion_traits*.
Thus, the *result_type* is well defined if *NT1* and *NT2* are a model of *ExplicitInteroperable*.
The actual *div* is performed with the semantic of that type.

The function is defined if *result_type* is a model of the *EuclideanRing* concept.

#include <CGAL/number_utils.h>

template< class NT1, class NT2>
result_type *div(NT1 x, NT2 y)*

See Also

EuclideanRing . . . . page 73
AlgebraicStructureTraits::Div . . . . page 93
CGAL::mod . . . . page 118
CGAL::div_mod . . . . page 116

Function

CGAL::mod

Definition

The function *mod* computes the remainder of division with remainder.

In case the argument types *NT1* and *NT2* differ, the *result_type* is determined via *Coercion_traits*.
Thus, the *result_type* is well defined if *NT1* and *NT2* are a model of *ExplicitInteroperable*.
The actual *mod* is performed with the semantic of that type.

The function is defined if *result_type* is a model of the *EuclideanRing* concept.

#include <CGAL/number_utils.h>

template< class NT1, class NT2>
result_type *mod(NT1 x, NT2 y)*

See Also

EuclideanRing . . . . page 73
AlgebraicStructureTraits::DivMod . . . . page 91
CGAL::div_mod . . . . page 116
CGAL::div . . . . page 117

CGAL::inverse

Function

Definition

The function *inverse* returns the inverse element with respect to multiplication.

The function is defined if the argument type is a model of the *Field* concept.

#include <CGAL/number_utils.h>

template <class NT>
NT *inverse(NT x)* *Precondition*: $x \neq 0$

See Also

Field . . . . page 74
AlgebraicStructureTraits::Inverse . . . . page 95

CGAL::sqrt

Definition

The function *sqrt* returns the square root of a value.

The function is defined if the argument type is a model of the *FieldWithSqrt* concept.

#include <CGAL/number_utils.h>

template <class NT>
NT *sqrt(NT x)*

See Also

FieldWithSqrt . . . . page 75
AlgebraicStructureTraits::Sqrt . . . . page 96

CGAL::kth_root

Function

Definition

The function *kth_root* returns the k-th root of a value.

The function is defined if the second argument type is a model of the *FieldWithKthRoot* concept.

#include <CGAL/number_utils.h>

template <class NT>
NT *kth_root(int k, NT x)*

See Also

FieldWithKthRoot . . . . page 76
AlgebraicStructureTraits::KthRoot . . . . page 97

Function

CGAL::root_of

Definition

The function *root_of* computes a real root of a square-free univariate polynomial.

The function is defined if the value type, *NT*, of the iterator range is a model of the *FieldWithRootOf* concept.

#include <CGAL/number_utils.h>

template <class InputIterator>
NT *root_of(int k, InputIterator begin, InputIterator end)*

returns the k-th real root of the univariate polynomial, which is defined by the iterator range, where begin refers to the constant term.
Precondition: The polynomial is square-free.

See Also

FieldWithRootOf . . . . page 77
AlgebraicStructureTraits::RootOf . . . . page 98

RealEmbeddable

Definition

A model of this concepts represents numbers that are embeddable on the real axis. The type obeys the algebraic structure and compares two values according to the total order of the real numbers.

Moreover, *CGAL::Real_embeddable_traits< RealEmbeddable >* is a model of *RealEmbeddableTraits*
with:
- *CGAL::Real_embeddable_traits< RealEmbeddable >::Is_real_embeddable* set to *Tag_true*
and functors :
- *CGAL::Real_embeddable_traits< RealEmbeddable >::Is_zero*
- *CGAL::Real_embeddable_traits< RealEmbeddable >::Abs*
- *CGAL::Real_embeddable_traits< RealEmbeddable >::Sgn*
- *CGAL::Real_embeddable_traits< RealEmbeddable >::Is_positive*
- *CGAL::Real_embeddable_traits< RealEmbeddable >::Is_negative*
- *CGAL::Real_embeddable_traits< RealEmbeddable >::Compare*
- *CGAL::Real_embeddable_traits< RealEmbeddable >::To_double*
- *CGAL::Real_embeddable_traits< RealEmbeddable >::To_interval*

Remark:
If a number type is a model of both *IntegralDomainWithoutDivision* and *RealEmbeddable*, it follows that the ring represented by such a number type is a sub-ring of the real numbers and hence has characteristic zero.

Refines

Equality Comparable
LessThanComparable

Operations

bool	$a == b$
bool	$a\, != b$
bool	$a < b$
bool	$a <= b$
bool	$a > b$
bool	$a >= b$

See Also

RealEmbeddableTraits . . . . page 124

Concept

RealEmbeddableTraits

Definition

A model of *RealEmbeddableTraits* is associated to a number type *Type* and reflects the properties of this type with respect to the concept *RealEmbeddable*.

Types

A model of *RealEmbeddableTraits* is supposed to provide:

RealEmbeddableTraits:: Type	The associated number type.
RealEmbeddableTraits:: Is_real_embeddable	Tag indicating whether the associated type is real embeddable. This is either *CGAL::Tag_true* or *CGAL::Tag_false*.
RealEmbeddableTraits:: Boolean	This type specifies the return type of the predicates provided by this traits. The type must be convertible to *bool* and typically the type indeed maps to *bool*. However, there are also cases such as interval arithmetic, in which it is *Uncertain< bool>* or some similar type.
RealEmbeddableTraits:: Sign	This type specifies the return type of the *Sgn* functor. The type must be convertible to *CGAL::Sign* and typically the type indeed maps to *CGAL::Sign*. However, there are also cases such as interval arithmetic, in which it is *Uncertain< CGAL::Sign>* or some similar type.
RealEmbeddableTraits:: Comparison_result	This type specifies the return type of the *Compare* functor. The type must be convertible to *CGAL::Comparison_result* and typically the type indeed maps to *CGAL::Comparison_result*. However, there are also cases such as interval arithmetic, in which it is *Uncertain<CGAL::Comparison_result>* or some similar type.

Functors

In case the associated type is *RealEmbeddable* all functors are provided.
In case a functor is not provided, it is set to *CGAL::Null_functor*.

RealEmbeddableTraits:: Is_zero	A model of *RealEmbeddableTraits::IsZero* In case *Type* is also model of *IntegralDomainWithoutDivision* this is a model of *AlgebraicStructureTraits::IsZero*.
RealEmbeddableTraits:: Abs	A model of *RealEmbeddableTraits::Abs*

RealEmbeddableTraits:: Sgn	A model of *RealEmbeddableTraits::Sgn*
RealEmbeddableTraits:: Is_positive	A model of *RealEmbeddableTraits::IsPositive*
RealEmbeddableTraits:: Is_negative	A model of *RealEmbeddableTraits::IsNegative*
RealEmbeddableTraits:: Compare	A model of *RealEmbeddableTraits::Compare*
RealEmbeddableTraits:: To_double	A model of *RealEmbeddableTraits::ToDouble*
RealEmbeddableTraits:: To_interval	A model of *RealEmbeddableTraits::ToInterval*

Has Models

CGAL::Real_embeddable_traits<T> . . . . page 134

RealEmbeddableTraits::IsZero

Definition

AdaptableUnaryFunction, returns true in case the argument is 0.

Refines

AdaptableUnaryFunction

Types

RealEmbeddableTraits::IsZero:: result_type Type convertible to *bool*.

RealEmbeddableTraits::IsZero:: argument_type

Is *RealEmbeddableTraits::Type*.

Operations

result_type *is_zero(argument_type x)*

returns true in case *x* is the zero element of the ring.

See Also

RealEmbeddableTraits . . . . page 124 *AlgebraicStructureTraits::IsZero* . . . . page 82

RealEmbeddableTraits::Abs

Definition

AdaptableUnaryFunction computes the absolute value of a number.

Refines

AdaptableUnaryFunction

Types

RealEmbeddableTraits::Abs:: result_type	Is *RealEmbeddableTraits::Type*.
RealEmbeddableTraits::Abs:: argument_type	Is *RealEmbeddableTraits::Type*.

Operations

result_type *abs(argument_type x)*

computes the absolute value of x.

See Also

RealEmbeddableTraits . . . . page 124

Functor
Concept

Functor Concept

RealEmbeddableTraits::Sgn

Definition

This *AdaptableUnaryFunction* computes the sign of a real embeddable number.

Refines

AdaptableUnaryFunction

Types

RealEmbeddableTraits::Sgn:: result_type	Type convertible to *CGAL::Sign*.
RealEmbeddableTraits::Sgn:: argument_type	Is *RealEmbeddableTraits::Type*.

Operations

result_type *sign(argument_type x)*

Computes the sign of *x*.

See Also

RealEmbeddableTraits . . . . page 124

RealEmbeddableTraits::IsPositive

Definition

AdaptableUnaryFunction, returns true in case the argument is positive.

Refines

AdaptableUnaryFunction

Types

RealEmbeddableTraits::IsPositive:: result_type

Type convertible to *bool*.

RealEmbeddableTraits::IsPositive:: argument_type

Is *RealEmbeddableTraits::Type*.

Operations

result_type *is_positive(argument_type x)*

returns true in case *x* is positive.

See Also

RealEmbeddableTraits . . . . page 124

RealEmbeddableTraits::IsNegative

Definition

AdaptableUnaryFunction, returns true in case the argument is negative.

Refines

AdaptableUnaryFunction

Types

RealEmbeddableTraits::IsNegative:: result_type

Type convertible to *bool*.

RealEmbeddableTraits::IsNegative:: argument_type

Is *RealEmbeddableTraits::Type*.

Operations

result_type   *is_negative(argument_type x)*

returns true in case x is negative.

See Also

RealEmbeddableTraits . . . . page 124

RealEmbeddableTraits::Compare

Functor
Concept

Definition

AdaptableBinaryFunction compares two real embeddable numbers.

Refines

AdaptableBinaryFunction

Types

RealEmbeddableTraits::Compare:: result_type

Type convertible to *CGAL::Comparison_result*.

RealEmbeddableTraits::Compare:: first_argument_type

Is *RealEmbeddableTraits::Type*.

RealEmbeddableTraits::Compare:: second_argument_type

Is *RealEmbeddableTraits::Type*.

Operations

result_type *compare(first_argument_type x, second_argument_type y)*

compares *x* with respect to *y*.

template <class NT1, class NT2>
result_type *compare(NT1 x, NT2 y)*

This operator is defined if *NT1* and *NT2* are *ExplicitInteroperable* with coercion type *RealEmbeddableTraits::Type*.

See Also

RealEmbeddableTraits . . . . page 124

RealEmbeddableTraits::ToDouble

Definition

AdaptableUnaryFunction computes a double approximation of a real embeddable number.

Remark: In order to control the quality of approximation one has to resort to methods that are specific to NT. There are no general guarantees whatsoever.

Refines

AdaptableUnaryFunction

Types

RealEmbeddableTraits::ToDouble:: result_type

Is *double*.

RealEmbeddableTraits::ToDouble:: argument_type

Is *RealEmbeddableTraits::Type*.

Operations

result_type *to_double(argument_type x)*

computes a double approximation of a real embeddable number.

See Also

RealEmbeddableTraits . . . . page 124

RealEmbeddableTraits::ToInterval

Definition

AdaptableUnaryFunction computes for a given real embeddable number x a double interval containing x. This interval is represented by *std::pair<double,double>*.

Refines

AdaptableUnaryFunction

Types

RealEmbeddableTraits::ToInterval:: result_type

Is *std::pair<double,double>*.

RealEmbeddableTraits::ToInterval:: argument_type

Is *RealEmbeddableTraits::Type*.

Operations

result_type *to_interval(argument_type x)*

computes a double interval containing x.

See Also

RealEmbeddableTraits . . . . page 124

Class

CGAL::Real_embeddable_traits<T>

Definition

An instance of *Real_embeddable_traits<T>* is a model of *RealEmbeddableTraits*, where T is the associated type.

#include <CGAL/Real_embeddable_traits.h>

Is Model for the Concepts

RealEmbeddableTraits . . . . page 124

CGAL::abs

Function

Definition

The template function *abs* returns the absolute value of a number.

The function is defined if the argument type is a model of the *RealEmbeddable* concept.

#include <CGAL/number_utils.h>

template <class NT>
NT *abs(NT x)*

See Also

RealEmbeddable . . . . page 123
RealEmbeddableTraits::Abs . . . . page 127

Function

CGAL::sign

Definition

The template function *sign* returns the sign of a number.

The function is defined if the argument type is a model of the *RealEmbeddable* concept.

#include <CGAL/number_utils.h>

template <class NT>

result_type	*sign(NT x)*	returns the sign of the argument. The *result_type* is convertible to *CGAL::Sign*.

See Also

RealEmbeddable . . . . page 123
RealEmbeddableTraits::Sgn . . . . page 128

CGAL::is_positive

Definition

The template function *is_positive* determines if a value is positive or not. The function is defined if the argument type is a model of the *RealEmbeddable* concept.

#include <CGAL/number_utils.h>

result_type	*is_positive(NT x)*	The *result_type* is convertible to *bool*.

See Also

RealEmbeddable . . . . page 123
RealEmbeddableTraits::IsPositive . . . . page 129

CGAL::is_negative

Definition

The template function *is_negative* determines if a value is negative or not. The function is defined if the argument type is a model of the *RealEmbeddable* concept.

#include <CGAL/number_utils.h>

result_type	*is_negative(NT x)*	The *result_type* is convertible to *bool*.

See Also

RealEmbeddable . . . . page 123
RealEmbeddableTraits::IsNegative . . . . page 130

CGAL::compare

Definition

The template function *compare* compares the first argument with respect to the second, i.e. it returns *CGAL::LARGER* if *x* is larger then *y*.

In case the argument types *NT1* and *NT2* differ, *compare* is performed with the semantic of the type determined via *Coercion_traits*. The function is defined if this type is a model of the *RealEmbeddable* concept.

#include <CGAL/number_utils.h>

template <class NT1, class NT2>

result_type	*compare(NT x, NT y)*	The *result_type* is convertible to *CGAL::Comparison_result*.

RealEmbeddable . . . . page 123
RealEmbeddableTraits::Compare . . . . page 131

Function

CGAL::to_double

Definition

The template function *to_double* returns an double approximation of a number. The function is defined if the argument type is a model of the *RealEmbeddable* concept.

Remark: In order to control the quality of approximation one has to resort to methods that are specific to NT. There are no general guarantees whatsoever.

#include <CGAL/number_utils.h>

template <class NT>
double *to_double(NT x)*

See Also

RealEmbeddable . . . . . . page 123
RealEmbeddableTraits::ToDouble . . . . . . page 132

CGAL::to_interval

Function

Definition

The template function *to_interval* computes for a given real embeddable number *x* a double interval containing *x*. This interval is represented by a *std::pair<double,double>*. The function is defined if the argument type is a model of the *RealEmbeddable* concept.

#include <CGAL/number_utils.h>

template <class NT>
std::pair<double,double>

to_interval(NT x)

See Also

RealEmbeddable . . . . page 123
RealEmbeddableTraits::ToInterval . . . . page 133

Concept

Fraction

Definition

A type is considered as a *Fraction*, if there is a reasonable way to decompose it into a numerator and denominator. In this case the relevant functionality for decomposing and re-composing as well as the numerator and denominator type are provided by *CGAL::Fraction_traits*.

See Also

FractionTraits . . . . page 143
CGAL::Fraction_traits<T> . . . . page 147

FractionTraits

Concept

Definition

A model of FractionTraits is associated with a type *Type*.

In case the associated type is a *Fraction*, a model of FractionTraits provides the relevant functionality for decomposing and re-composing as well as the numerator and denominator type.

#include <CGAL/Fraction_traits.h>

Types

FractionTraits:: Type	the associated type
FractionTraits:: Is_fraction	Tag indicating whether the associated type is a fraction and can be decomposed into a numerator and denominator. This is either *CGAL::Tag_true* or *CGAL::Tag_false*.
FractionTraits:: Numerator_type	The type to represent the numerator. This is undefined in case the associated type is not a fraction.
FractionTraits:: Denominator_type	The (simpler) type to represent the denominator. This is undefined in case the associated type is not a fraction.

Functors

In case *Type* is not a *Fraction* all functors are *Null_functor*.

FractionTraits:: Decompose	A model of *FractionTraits::Decompose*.
FractionTraits:: Compose	A model of *FractionTraits::Compose*.
FractionTraits:: Common_factor	A model of *FractionTraits::CommonFactor*.

Has Models

CGAL::Fraction_traits<T> . . . . page 147

See Also

FractionTraits::Decompose . . . . page 144
FractionTraits::Compose . . . . page 145
FractionTraits::CommonFactor . . . . page 146

FractionTraits::Decompose

Definition

Functor decomposing a *Fraction* into its numerator and denominator.

Types

Operations

void *decompose.operator()(FractionTraits::Type f,*
FractionTraits::Numerator_type `&` *n,*
FractionTraits::Denominator_type `&` *d)*

decompose f into numerator n and denominator d.

See Also

Fraction . . . page 142
FractionTraits . . . page 143
FractionTraits::Compose . . . page 145
FractionTraits::CommonFactor . . . page 146

FractionTraits::Compose

Functor
Concept

Definition

AdaptableBinaryFunction, returns the fraction of its arguments.

Refines

AdaptableBinaryFunction

Types

typedef FractionTraits::Type

result_type;

typedef FractionTraits::Numerator_type

first_argument_type;

typedef FractionTraits::Denominator_type

second_argument_type;

Operations

result_type *compose(first_argument_type n, second_argument_type d)*

return the fraction n/d.

See Also

Fraction . . . . page 142
FractionTraits . . . . page 143
FractionTraits::Decompose . . . . page 144
FractionTraits::CommonFactor . . . . page 146

FractionTraits::CommonFactor

Definition

AdaptableBinaryFunction, finds great common factor of denominators.

This can be considered as a relaxed version of *AlgebraicStructureTraits::Gcd*, this is needed because it is not guaranteed that *FractionTraits::Denominator_type* is a model of *UniqueFactorizationDomain*.

Refines

AdaptableBinaryFunction

Types

typedef FractionTraits::Denominator_type

result_type;

typedef FractionTraits::Denominator_type

first_argument_type;

typedef FractionTraits::Denominator_type

second_argument_type;

Operations

result_type *common_factor(first_argument_type d1, second_argument_type d2)*

return a great common factor of $d1$ and $d2$.
Note: `operator()(0,0) = 0`

See Also

CGAL::Fraction_traits<T>

Class

Definition

An instance of *Fraction_traits<T>* is a model of *FractionTraits*, where *T* is the associated type.

#include <CGAL/Fraction_traits.h>

Is Model for the Concepts

FractionTraits . . . . page 143

RingNumberType

Definition

The concept RingNumberType combines the requirements of the concepts *IntegralDomainWithoutDivision* and *RealEmbeddable*. A model of RingNumberType can be used as a template parameter for Homogeneous kernels.

Refines

IntegralDomainWithoutDivision
RealEmbeddable

Has Models

C++ built-in number types
CGAL::Gmpq
CGAL::Gmpz
CGAL::Interval_nt
CGAL::Interval_nt_advanced
CGAL::Lazy_exact_nt<RingNumberType>
CGAL::MP_Float
CGAL::Gmpzf
CGAL::Quotient<RingNumberType>
leda_integer
leda_rational
leda_bigfloat
leda_real

See Also

FieldNumberType . . . . page 149

FieldNumberType

Concept

Definition

The concept FieldNumberType combines the requirements of the concepts *Field* and *RealEmbeddable*. A model of FieldNumberType can be used as a template parameter for Cartesian kernels.

Refines

Field
RealEmbeddable

Has Models

float
double
CGAL::Gmpq
CGAL::Interval_nt
CGAL::Interval_nt_advanced
CGAL::Lazy_exact_nt<FieldNumberType>
CGAL::Quotient<RingNumberType>
leda_rational
leda_bigfloat
leda_real

See Also

RingNumberType . . . . page 148
Kernel . . . . page 581

Chapter 5

Number Types

Michael Hemmer, Susan Hert, Sylvain Pion, and Stefan Schirra

Contents

5.1 Introduction

This chapter gives an overview of the number types supported by CGAL. Number types must fulfill certain syntactical and semantic requirements, such that they can be successfully used in CGAL code. In general they are expected to be a model of an algebraic structure concepts and in case they model a subring of the real numbers they are also a model of *RealEmbeddable*. For an overview of the algebraic structure concepts see Section 4.8.

5.2 Built-in Number Types

The built-in number types *float*, *double* and *long double* have the required arithmetic and comparison operators. They lack some required routines though which are automatically included by CGAL. [1]

[1] The functions can be found in the header files `CGAL/int.h`, `CGAL/float.h`, `CGAL/double.h` and `CGAL/long_long.h`.

All built-in number types of C++ can represent a discrete (bounded) subset of the rational numbers only. We assume that the floating-point arithmetic of your machine follows IEEE floating-point standard. Since the floating-point culture has much more infrastructural support (hardware, language definition and compiler) than exact computation, it is very efficient. Like with all number types with finite precision representation which are used as approximations to the infinite ranges of integers or real numbers, the built-in number types are inherently potentially inexact. Be aware of this if you decide to use the efficient built-in number types: you have to cope with numerical problems. For example, you can compute the intersection point of two lines and then check whether this point lies on the two lines. With floating point arithmetic, roundoff errors may cause the answer of the check to be *false*. With the built-in integer types overflow might occur.

5.3 Number Types Provided by CGAL

CGAL provides several number types that can be used for exact computation. These include the *Quotient* class that can be used to create, for example, a number type that behaves like a rational number when parameterized with a number type which can represent integers.

The number type *MP_Float* is able to represent multi-precision floating point values, a generalization of integers scaled by a (potentially negative) power of 2. It allows to deal with ring operations over floating-point values with requiring rational numbers. By plugging it in *Quotient*, one obtains rational numbers. Note that *MP_Float* may not be as efficient as the integer types provided by GMP or LEDA, but it has the advantage to make more parts of CGAL independent on these external libraries for handling robustness issues.

The templated number type *Lazy_exact_nt<NT>* is able to represent any number that *NT* is able to represent, but because it first tries to use an approximate value to perform computations it can be faster than *NT*.

A number type for doing interval arithmetic, *Interval_nt*, is provided. This number type helps in doing arithmetic filtering in many places such as *Filtered_predicate*.

CGAL::Sqrt_extension is a number type that allows to represent algebraic numbers of degree 2 as well as nested forms. A generic function *CGAL::make_root_of_2* allows to build this type generically.

A debugging helper *Number_type_checker<NT1,NT2,Comparator>* is also provided which allows to compare the behavior of operations over two number types.

5.4 Number Types Provided by GMP

CGAL provides wrapper classes for number types defined in the GNU Multiple Precision arithmetic library [Gra]. The file `CGAL/Gmpz.h` provides the class *Gmpz*, a wrapper class for the arbitrary-precision integer type *mpz_t*, which is compliant with the CGAL number type requirements. The file `CGAL/Gmpq.h` provides the class *Gmpq*, a wrapper class for the arbitrary-precision rational type *mpq_t*, which is compliant with the CGAL number type requirements.

The file `CGAL/Gmpzf.h` provides the class *Gmpzf*, an exact arbitrary-precision floating-point type. Hence, It does not support operators like `/` to guarantee exactness of the operations. The arithmetic operations on this type are restricted to `+`, `-`, `*` and *CGAL::integral_division*.

The file `CGAL/Gmpfr.h` provides the class *Gmpfr*, a fixed-precision floating-point number type. Since the precision (number of bits used to represent the mantissa of the number) is fixed for each object, the result of each operation is rounded when necessary. Though not necessary at first, the user will take full advantage of this number type by understanding the ideas behind floating-point arithmetic, such as precision and rounding, and understanding the flags set by this library after each operation. For more details, the reader should refer to [MPFb] and the *Gmpfr* reference manual.

In addition, it is possible to directly use the C++ number types provided by GMP : *mpz_class*, *mpq_class* (note that support for *mpf_class* is incomplete). The file `CGAL/gmpxx.h` provides the necessary functions to make these classes compliant to the CGAL number type requirements.

To use these classes, GMP and MPFR must be installed.

5.5 Number Types Provided by LEDA

LEDA provides number types that can be used for exact computation with both Cartesian and homogeneous representations. If you are using homogeneous representation with the built-in integer types *short*, *int*, and *long* as ring type, exactness of computations can be guaranteed only if your input data come from a sufficiently small integral range and the depth of the computations is sufficiently small. LEDA provides the number type *leda_integer* for integers of arbitrary length. (Of course the length is somehow bounded by the resources of your computer.) It can be used as ring type in homogeneous kernels and leads to exact computation as long as all intermediate results are rational. For the same kind of problems, Cartesian representation with number type *leda_rational* leads to exact computation as well. The number type *leda_bigfloat* in LEDA is a variable precision floating-point type. Rounding mode and precision (i.e. mantissa length) of *leda_bigfloat* can be set.

The most sophisticated number type in LEDA is the number type called *leda_real*. Like in Pascal, where the name *real* is used for floating-point numbers, the name *leda_real* does not describe the number type precisely, but intentionally. *leda_real*s are a subset of real algebraic numbers. Any integer is *leda_real* and *leda_real*s are closed under the operations $+,-,*,/$ and k-th root computation. For LEDA version 5.0 and or later *leda_real* is also able to represent real roots of polynomials. *leda_real*s guarantee that all comparisons between expressions involving *leda_real*s produce the exact result.

The files `CGAL/leda_integer.h`, `CGAL/leda_rational.h`, `CGAL/leda_bigfloat.h` and `CGAL/leda_real.h` provide the necessary functions to make these classes compliant to the CGAL number type requirements.

5.6 Number Types Provided by CORE

In principle CORE [KLPY99] provides the same set of number types as LEDA. The type *CORE::BigInt* represent integers and *CORE::BigRat* represent rationals of arbitrary length. The number type *CORE::BigFloat* is a variable precision floating-point type. It is also possible to interpret it as an interval type, since it also carries the error of a computed value. As for LEDA, the most sophisticated number type in CORE is *CORE::Expr*, which is in its functionality equivalent to *leda_real*.

The files `CGAL/CORE_BigInt.h`, `CGAL/CORE_BigRat.h`, `CGAL/CORE_BigFloat.h` and `CGAL/CORE_Expr.h` provide the necessary functions to make these classes compliant to the CGAL number type requirements.

CORE version 1.7 or later is required.

5.7 Interval arithmetic

Interval arithmetic is very important for geometric programming. It is a fundamental tool for filtering predicates. For many problems, intervals of machine double-precision numbers are sufficient, but it is not always enough. For example, one approach for determining the sign of an expression is to evaluate its sign using interval arithmetic and to repeatedly increase the precision of the bounds of the intervals until either the interval does not contain zero or its width is less than the separation bound of the polynomial.

For intervals of machine double-precision numbers, CGAL provides the class *Interval_nt*. For intervals of floating-point arbitrary-precision numbers, the class *Gmpfi* is provided in the file `CGAL/Gmpfi.h`.

Endpoints of *Gmpfi* intervals are represented as *Gmpfr* numbers. Each interval has an associated *precision*, which is the maximum precision (number of bits used to represent the mantissa) of its endpoints. The result of the operations is guaranteed to be always contained in the returned interval. Since the interval arithmetic is implemented on top of *Gmpfr*, the global flags and the default precision are inherited from the *Gmpfr* interface. See [MPFa] and the *Gmpfi* reference manual for details.

To use the *Gmpfi* class, MPFI must be installed.

5.8 User-supplied Number Types

In order to use your own number type it must be a model of the according algebraic structure concept, in particular you must provide a specialization of *CGAL::Algebraic_structure_traits* and also of *Real_embeddable_traits* in case it is a sub ring of the real numbers. If you even want to provide a related ensemble of number types you should also provide specializations for *CGAL::Coercion_traits* in order to reflect their interoperability.

5.9 Design and Implementation History

This package was naturally one of the first packages implemented in CGAL. It initially contained the *Quotient*, *Gmpz* and *Gmpq* classes, together with the interfaces to the number types provided by LEDA, which were implemented by Stefan Schirra and Andreas Fabri.

Later, around 1998-2002, Sylvain Pion implemented *Interval_nt MP_Float* and *Lazy_exact_nt*, together with the interfaces to the *mpz_class* and *mpq_class* types from GMP.

Number type concepts were then refined, notably by Lutz Kettner and Susan Hert, who also contributed utility algorithms.

The work on concepts was further extended within the EXACUS project, and was finally contributed to CGAL by Michael Hemmer in 2006, as what is now the separate Algebraic Foundations package 4, together with a rewritten interface to operations on number types.

The class *Sqrt_extension* was contributed by Michael Hemmer and Ron Wein around 2006. In 2010 it went through a considerable reinvestigation by Sébastien Loriot, Michael Hemmer, and Monique Teillaud. As a result it got further improved and now replaces several similar types such as *Root_of_2*, which had been contributed by Pedro M. M. de Castro, Sylvain Pion and Monique Teillaud, and is deprecated since CGAL-3.8.

In 2008-2010, Bernd Gärtner added the *Gmpzf* class, while Luis Peñaranda and Sylvain Lazard contributed the *Gmpfi* and *Gmpfr* classes.

Number Types
Reference Manual

Michael Hemmer, Susan Hert, Sylvain Pion, and Stefan Schirra

5.10 Classified Reference Pages

Number Type Classes and Concepts

CORE

LEDA

GMP

CGAL

Relates Rational

Relates Algebraic Extensions

Utilities

5.11 Alphabetical List of Reference Pages

Class

int

Definition

The fundamental type *int* is an *RealEmbeddable EuclideanRing*. Due to overflow *int* is considered as not exact.

#*include* <*CGAL/int.h*> **Is Model for the Concepts**

EuclideanRing
RealEmbeddable

short int

Class

Definition

The fundamental type *short int* is an *RealEmbeddable EuclideanRing*. Due to overflow *short int* is considered as not exact.

#include <CGAL/int.h> **Is Model for the Concepts**

EuclideanRing
RealEmbeddable

Class

long int

Definition

The fundamental type *long int* is an *RealEmbeddable EuclideanRing*. Due to overflow *long int* is considered as not exact.

#*include* <*CGAL/int.h*> **Is Model for the Concepts**

EuclideanRing
RealEmbeddable

long long int

Class

Definition

The fundamental type *long long int* is an *RealEmbeddable EuclideanRing*. Due to overflow *long long int* is considered as not exact.

#include <*CGAL/long_long.h*> **Is Model for the Concepts**

EuclideanRing
RealEmbeddable

Class

float

Definition

The fundamental type *float* is an *RealEmbeddable FieldWithSqrt*. Due to rounding errors and overflow *float* is considered as not exact.

#include <CGAL/float.h> **Is Model for the Concepts**

FieldWithSqrt
RealEmbeddable

Query Functions

bool	*CGAL::is_finite(x)*	Determines whether the argument represents a value in $\mathbb{R}$.

double

Class

Definition

The fundamental type *double* is an *RealEmbeddable Field*. Due to rounding errors and overflow *double* is considered as not exact.

#include <CGAL/double.h> **Is Model for the Concepts**

FieldWithSqrt
RealEmbeddable

Query Functions

bool	*CGAL::is_finite(x)*	Determines whether the argument represents a value in $\mathbb{R}$.

Class

long double

Definition

The fundamental type *long double* is an *RealEmbeddable FieldWithSqrt*. Due to rounding errors and overflow *long double* is considered as not exact.

#include <CGAL/long_double.h> **Is Model for the Concepts**

FieldWithSqrt
RealEmbeddable

Query Functions

bool	*CGAL::is_finite(x)*	Determines whether the argument represents a value in $\mathbb{R}$.

CORE::BigInt

Definition

The class *CORE::BigInt* provides exact computation in $\mathbb{Z}$. Operations and comparisons between objects of this type are guaranteed to be exact. This number type is provided by the CORE library [KLPY99].

CGAL defines the necessary functions so that this class complies to the requirements on number types.

#include <CGAL/CORE_BigInt.h>

Is Model for the Concepts

EuclideanRing

RealEmbeddable

CORE::BigRat

Definition

The class *CORE::BigRat* provides exact computation in $\mathbb{R}$. Operations and comparisons between objects of this type are guaranteed to be exact. This number type is provided by the CORE library [KLPY99].

CGAL defines the necessary functions so that this class complies to the requirements on number types.

#include <CGAL/CORE_BigRat.h>

Is Model for the Concepts

Field
RealEmbeddable
Fraction
FromDoubleConstructible

CORE::BigFloat

Class

Definition

The class *CORE::BigFloat* is a variable precision floating-point type. Rounding mode and precision (i.e. mantissa length) of *CORE::BigFloat* can be set. Since it also carries the error of a computed value.

This number type is provided by the CORE library [KLPY99].

CGAL defines the necessary functions so that this class complies to the requirements on number types.

#include <CGAL/CORE_BigFloat.h>

Is Model for the Concepts

FieldWithKthRoot
RealEmbeddable
FromDoubleConstructible

Class

CORE::Expr

Definition

The class *CORE::Expr* provides exact computation over the subset of real numbers that contains integers, and which is closed by the operations $+, -, \times, /, \sqrt{}$ and *kth_root*. Operations and comparisons between objects of this type are guaranteed to be exact. This number type is provided by the CORE library [KLPY99].

CGAL defines the necessary functions so that this class complies to the requirements on number types.

#include <CGAL/CORE_Expr.h>

Is Model for the Concepts

FieldWithRootOf
RealEmbeddable
FromDoubleConstructible

leda_integer

Definition

The class *leda_integer* provides exact computation in $\mathbb{Z}$. The class *leda_integer* is a wrapper class that provides the functions needed to use the number type *leda::integer*, representing exact multiprecision integers provided by LEDA.

#include <CGAL/leda_integer.h>

Is Model for the Concepts

EuclideanRing

RealEmbeddable

For more details on the number types of LEDA we refer to the LEDA manual [MNSU].

Class

leda_rational

Definition

The class *leda_rational* provides exact computation in $\mathbb{R}$. The class *leda_rational* is a wrapper class that provides the functions needed to use the number type *rational*, representing exact multiprecision rational numbers provided by LEDA.

#include <CGAL/leda_rational.h>

Is Model for the Concepts

Field
RealEmbeddable
Fraction
FromDoubleConstructible

For more details on the number types of LEDA we refer to the LEDA manual [MNSU].

leda_bigfloat

Class

Definition

The class *leda_bigfloat* is a wrapper class that provides the functions needed to use the number type *bigfloat*. *bigfloat* Rounding mode and precision (i.e. mantissa length) of *leda_bigfloat* can be set.

For more details on the number types of LEDA we refer to the LEDA manual [MNSU].

#include <CGAL/leda_bigfloat.h>

Is Model for the Concepts

FieldWithKthRoot
RealEmbeddable
FromDoubleConstructible

Class

leda_real

Definition

The class *leda_real* is a wrapper class that provides the functions needed to use the number type *real*, representing exact real numbers numbers provided by LEDA. The class *leda_real* provides exact computation over the subset of real numbers that contains integers, and which is closed by the operations $+, -, \times, /, \sqrt{}$ and *kth_root*. For LEDA version 5.0 or later *leda_real* is also able to represent real roots of polynomials. Operations and comparisons between objects of this type are guaranteed to be exact.

#include <CGAL/leda_real.h>

Is Model for the Concepts

FieldWithRootOf
RealEmbeddable
FromDoubleConstructible

For more details on the number types of LEDA we refer to the LEDA manual [MNSU].

mpz_class

Class

Definition

The class *mpz_class* is an exact multiprecision integer number type, provided by GMP. CGAL provides the necessary functions to make it compliant to the number type concept.

#include <CGAL/gmpxx.h>

Is Model for the Concepts

EuclideanRing

RealEmbeddable

See the GMP documentation for additional details.

Class

mpq_class

Definition

The class *mpq_class* is an exact multiprecision rational number type, provided by GMP. CGAL provides the necessary functions to make it compliant to the number type concept.

#include <CGAL/gmpxx.h>

Is Model for the Concepts

Field
RealEmbeddable
Fraction

See the GMP documentation for additional details.

CGAL::Gmpz

Class

Definition

An object of the class *Gmpz* is an arbitrary precision integer based on the GNU Multiple Precision Arithmetic Library.

#include <CGAL/Gmpz.h>

Is Model for the Concepts

EuclideanRing
RealEmbeddable

Creation

Gmpz z;	creates an uninitialized multiple precision integer z.
Gmpz z(int i);	creates a multiple-precision integer initialized with i.
Gmpz z(double d);	creates a multiple-precision integer initialized with the integral part of d.

Operations

Gmpz &	$++z$	prefix increment.
Gmpz	$z++$	postfix increment.
Gmpz &	$--z$	prefix decrement.
Gmpz	$z--$	postfix decrement.
Gmpz	$a >>$ *unsigned long i*	rightshift by i.
Gmpz	$a <<$ *unsigned long i*	leftshift by i.
Gmpz &	$z >>=$ *long i*	rightshift by i, where $i >= 0$.
Gmpz &	$z <<=$ *long i*	leftshift by i, where $i >= 0$.
Gmpz	$a \,\&\, b$	bitwise AND.
Gmpz	$a \mid b$	bitwise IOR.
Gmpz	$a\,\hat{}\,b$	bitwise XOR.
Gmpz &	$z \,\&= b$	bitwise AND.
Gmpz &	$z \mid= b$	bitwise IOR.
Gmpz &	$z\,\hat{}= b$	bitwise XOR.
Sign	*z.sign() const*	Returns the sign of z.
size_t	*z.bit_size() const*	Returns the bit-size (that is, the number of bits needed to represent the mantissa) of z.
size_t	*z.size() const*	Returns the size in limbs of z. A limb is the type used by GMP to represent the integer (usually *long*).

size_t	*z.approximate_decimal_length() const*	
		Returns the approximate number of decimal digits needed to represent *z*. Approximate means either a correct result, either the correct result plus one.
double	*z.to_double() const*	Returns a double approximation of *z*. The integer is truncated if needed. If the exponent of the conversion is too big, the result is system dependent (returning infinity where it is supported).
std::ostream&	*std::ostream& out* << *z*	writes *z* to the ostream *out*.
std::istream&	*std::istream& in* >> & *z*	reads an integer from *in*, then converts it to a *Gmpz*.

Implementation

*Gmpz*s are reference counted.

CGAL::Gmpq

Class

Definition

An object of the class *Gmpq* is an arbitrary precision rational number based on the GNU Multiple Precision Arithmetic Library.

#include <CGAL/Gmpq.h>

Is Model for the Concepts

Field
RealEmbeddable
Fraction

Creation

Gmpq q;	creates an uninitialized *Gmpq* q.
Gmpq q(int i);	creates a *Gmpq* initialized with i.
Gmpq q(Gmpz n);	creates a *Gmpq* initialized with n.
Gmpq q(Gmpfr f);	creates a *Gmpq* initialized with f.
Gmpq q(int n, int d);	creates a *Gmpq* initialized with *n/d*.
Gmpq q(signed long n, unsigned long d);	creates a *Gmpq* initialized with *n/d*.
Gmpq q(unsigned long n, unsigned long d);	creates a *Gmpq* initialized with *n/d*.
Gmpq q(Gmpz n, Gmpz d);	creates a *Gmpq* initialized with *n/d*.
Gmpq q(double d);	creates a *Gmpq* initialized with d.
Gmpq q(std::string str);	creates a *Gmpq* initialized with *str*, which can be an integer like "41" or a fraction like "41/152". White space is allowed in the string, and ignored.
Gmpq q(std::string str, int base);	creates a *Gmpq* initialized with *str* in base *base*, which is an integer between 2 and 62. White space in the string is ignored.

Operations

std::ostream&	*std::ostream& out << q*	writes q to the ostream *out*, in the form n/d.
std::istream&	*std::istream& in >> & q*	reads a number from *in*, then converts it to a *Gmpq*. The number may be an integer, a rational number in the form n/d, or a floating-point number.

There are two access functions, namely to the numerator and the denominator of a rational. Note that these values are not uniquely defined. It is guaranteed that *q.numerator()* and *q.denominator()* return values *nt_num* and *nt_den* such that q = *nt_num/nt_den*, only if *q.numerator()* and *q.denominator()* are called consecutively wrt. q, i.e. q is not involved in any other operation between these calls.

Gmpz	*q.numerator() const*	returns the numerator of q.

Gmpz	*q.denominator() const*	returns the denominator of q.

Implementation

*Gmpq*s are reference counted.

CGAL::Gmpzf

Class

Definition

This is an multiple-precision floating-point type; it can represent numbers of the form $m * 2^e$, where m is an arbitrary precision integer based on the GNU Multiple Precision Arithmetic Library, and e is of type *long*. This type can be considered exact, even if the exponent is not a multiple-precision number. This number type offers functionality very similar to *MP_Float* but is generally faster.

#include <CGAL/Gmpzf.h>

Is Model for the Concepts

EuclideanRing

RealEmbeddable

Creation

Gmpzf f;	creates a *Gmpzf* initialized with 0.
Gmpzf f(int i);	creates a *Gmpzf* initialized with *i*.
Gmpzf f(long int l);	creates a *Gmpzf* initialized with *l*.
Gmpzf f(Gmpz i);	creates a *Gmpzf* initialized with *i*.
Gmpzf f(Gmpfr f);	creates a *Gmpzf* initialized with *f*.
Gmpzf f(double d);	creates a *Gmpzf* initialized with *d*.

Operations

std::ostream&	*std::ostream& out << f*	
		writes a double approximation of *f* to the ostream *out*.
std::ostream&	*print(std::ostream& out, f)*	
		writes an exact representation of *f* to the ostream *out*.
std::istream&	*std::istream& in >> & f*	
		reads a *double* from *in*, then converts it to a *Gmpzf*.

Implementation

The significand m of a *Gmpzf* is a *Gmpz* and is reference counted. The exponent e of a *Gmpzf* is a `long`.

CGAL::Gmpfr

Class

Definition

An object of the class *Gmpfr* is a fixed precision floating-point number, based on the MPFR library. This type is inexact, due to the fact that the mantissa of each number is represented by a fixed amount of bits (this amount is called *precision*). If an operation needs more bits than the precision of the result number, the results are rounded following different possible criteria (called *rounding modes*).

Currently, MPFR supports four rounding modes: round to nearest, round toward zero, round down (or toward $-\infty$) and round up (or toward $+\infty$). When not specified explicitly, the operations use the default rounding mode, which is in practice a variable local to each execution thread. The default rounding mode can be set to any of the four rounding modes (initially, it is set to nearest). To specify rounding modes for operations, the type used is *std::float_round_style*.

This type is *ImplicitInteroperable* with *Gmpz*, `long`, `unsigned long`, `int`, `double` and `long double`.

#include <CGAL/Gmpfr.h>

Is Model for the Concepts

FieldWithKthRoot
RealEmbeddable

Types

Gmpfr:: Precision_type	Type representing the precision (number of bits used to represent the mantissa) of a number.

Creation

Gmpfr f;	Creates an uninitialized *Gmpfr f*.
Gmpfr f(n);	Copy constructor. The copied object inherits the precision of *n*, and thus it is not rounded.
Gmpfr f(long si);	Creates *Gmpfr*, initialized with the value of *si*.
Gmpfr f(unsigned long ui);	Creates a *Gmpfr*, initialized with the value of *ui*.
Gmpfr f(int i);	Creates a *Gmpfr*, initialized with the value of *i*.
Gmpfr f(double d);	Creates a *Gmpfr*, initialized with the value of *d*.
Gmpfr f(long double ld);	Creates a *Gmpfr*, initialized with the value of *ld*.
Gmpfr f(Gmpz z);	Creates a *Gmpfr*, initialized with the value of *z*.

Gmpfr f(Gmpzf zf); — Creates a *Gmpfr*, initialized with the value of zf.

Gmpfr f(std::pair<Gmpz,long> ie); — Creates a *Gmpfr*, initialized with the value of $ie.first \times 2^{ie.second}$.

Note that all constructors can be called with two optional parameters. One can specify as second parameter the rounding mode desired for the conversion from the source number and as a third parameter the precision with which this *Gmpfr* will be created. If only one optional parameter is specified, it can be either the rounding mode or the precision. If no optional parameters are specified, the precision of the created object is chosen in such a way that the conversion is exact (i.e., no rounding is performed).

These optional parameters, along with other functions which will be explained below, allow users to control the rounding and precision. For example, being `z` a *Gmpz*, `Gmpfr g(z,53,std::round_toward_neg_infinity)` will construct a *Gmpfr* `g` having as value the biggest 53-bit floating-point number that is equal or smaller than to `z`.

Operations

Each Gmpfr object has a precision associated to it. The precision is the amount of bits needed to represent the mantissa. MPFR has a default precision value, which can be controlled by static functions of the Gmpfr class (in practice, this default value is a variable local to each execution thread). There are also functions to get and set the precision of each Gmpfr object.

static Precision_type *get_default_precision()* — This returns the current precision used in *Gmpfr* creation by default.

static Precision_type *set_default_precision(Precision_type p)*

This function sets the default MPFR precision to p, and returns the old one.

Precision_type *f.get_precision() const* — Returns the precision of f.

Gmpfr *f.round(Precision_type p, std::float_round_style r) const*

Returns the value of f, rounded with precision p in the direction r.

static std::float_round_style

get_default_rndmode() — This function returns the current rounding mode used by MPFR.

static std::float_round_style

set_default_rndmode(std::float_round_style r)

This function sets the MPFR rounding mode to r and returns the old one.

MPFR provides some flags to know whether performed operations were exact or not, or they incurred in overflow or underflow, if the exponent is out of range, or the result was *NaN* (not-a-number). One can clear the flags before a set of operations and inspect them afterward, in order to see if something unexpected happened during the operations. The static functions used to handle flags are:

static void	*clear_flags()*	Clears all the flags set by MPFR(they are not cleared automatically).
static bool	*underflow_flag()*	Shows whether an operation incurred in underflow.
static bool	*overflow_flag()*	Shows whether an operation incurred in overflow.
static bool	*nan_flag()*	Shows whether the result of an operation was *NaN*.
static bool	*inex_flag()*	Shows whether an operation was inexact.
static bool	*erange_flag()*	Returns *true* iff a range error occurred. Such an exception occurs when some function which does not return a *Gmpfr* has an invalid result. For example, this flag will be set if one of the operands of a comparison is *NaN*.

Arithmetic operators `+`, `-`, `*` and `/` are overloaded, but special care must be taken when applying them. The precision of an operation between two *Gmpfr*s is defined as the maximum of the operands precision and the default precision.

The second operand of the former operations can be a `Gmpfr`, `int`, `long`, `unsigned`, `unsigned long`, or `Gmpz`. The precision of an operation between a *Gmpfr* and a number of another type is defined as the maximum between the number's precision and the default precision.

To specify the rounding mode and/or the precision to perform an operation, this class provides the four static functions `add`, `sub`, `mul` and `div`. Only one of them is shown here, since their interfaces are similar:

static Gmpfr	*add(a, b)*
static Gmpfr	*add(a, b, std::float_round_style r)*
static Gmpfr	*add(a, b, Precision_type p)*
static Gmpfr	*add(a, b, Precision_type p, std::float_round_style r)*

When the precision is not specified in this family of functions, it is defined as in the overloaded operators. When the rounding mode is not specified, the default is used.

Other arithmetic functions provided by the class are:

Gmpfr *f.abs(Precision_type p, std::float_round_style r=get_default_rndmode())*

Returns the absolute value of f, rounded with precision p in the direction r. If p is not specified, the precision used is the maximum between f's precision and the default.

Gmpfr *f.sqrt(Precision_type p, std::float_round_style r=get_default_rndmode())*

Returns the square root of f, rounded with precision p in the direction r. If p is not specified, the precision used is the maximum between f's precision and the default.

Gmpfr *f.kthroot(int k, Precision_type p, std::float_round_style r=get_default_rndmode())*

Returns the k-th root of f, rounded with precision p in the direction r. If p is not specified, the precision used is the maximum between f's precision and the default.

Gmpfr *f.square(Precision_type p, std::float_round_style r=get_default_rndmode())*

Returns the square of f, rounded with precision p in the direction r. If p is not specified, the precision used is the maximum between f's precision and the default.

double *f.to_double(std::float_round_style r=get_default_rndmode())*

Returns a double precision approximation of f using the rounding mode r.

std::pair<double,double>

f.to_interval() Returns an interval of doubles which contains f. If a rounded endpoint does not fit in a double, the double is set to plus or minus infinity and the *overflow* or *underflow* flag.

std::pair<double,long>

f.to_double_exp(std::float_round_style r=get_default_rndmode())

Returns the pair (d,e) such that $0.5 \leq |d| < 1$ and $d \times 2^e$ equals f rounded to double precision, using the rounding mode r. If f is *NaN* or infinity, then the corresponding double is returned, leaving the exponent undefined and setting the appropriate error flag.

std::pair<std::pair<double,double>,long>

f.to_interval_exp() Returns $((m,M),e)$ such that $m \times 2^e \leq f \leq M \times 2^e$. If f is *NaN* or infinity, then the corresponding doubles are returned, leaving the exponent undefined and setting the appropriate error flag.

std::pair<Gmpz,long>

	f.to_integer_exp()	Returns a pair of integers (m,e), such that $f = m \times 2^e$. Note that the returned value of m is not necessarily the smallest possible value of m (that is, it might be that $2|m$).

Comparisons

Comparison operators `==`, `!=`, `>`, `<`, `>=` and `<=` are also overloaded. A *Gmpfr* can be compared with other *Gmpfr*, as well as with a *Gmpz*, `long`, `unsigned long`, `int`, `double` or `long double`. It is worth noting that the numbers are never converted nor rounded before comparison. In the case where one of the compared numbers is *NaN*, the *erange* flag is set.

Query Functions

Sign	*f.sign()*	Returns the sign of f.
bool	*f.is_zero()*	Returns *true* iff f is zero.
bool	*f.is_one()*	Returns *true* iff f is one.
bool	*f.is_nan()*	Returns *true* iff f is NaN (not-a-number).
bool	*f.is_inf()*	Returns *true* iff f is plus or minus infinity.
bool	*f.is_number()*	Returns *true* iff f is a valid number.
bool	*f.is_square()*	Returns *true* iff f is the square of a number representable by an object of this type.
bool	*f.is_square(y)*	Returns *true* iff f is the square of a number representable by an object of this type, computing and storing it in y.

Input/Output

std::istream&	*std::istream& in >> & f*	Reads a floating-point number from *in*. The number $M \times 2^E$ must be in the form MeE, where the mantissa M and the exponent E are integers in base 10.
std::ostream&	*std::ostream& out << f*	If the ostream *out* is in pretty-print mode, writes a decimal approximation of f to *out*. Otherwise, writes f to *out* in the form MeE, where M is its mantissa and E is its exponent, both in base 10.

Implementation

Since the MPFR library can be compiled to be thread-safe, this interface is designed to keep the thread-safety.

*Gmpfr*s are reference counted. This behavior may be changed, by setting the flag `CGAL_GMPFR_NO_REFCOUNT`. A non-reference-counted class is slightly more efficient in case the implementation does not need to copy numbers (this is not usually the case). Nevertheless, setting this flag may be useful for debugging purposes.

See Also

RealEmbeddable . . . . page 123
FieldWithKthRoot . . . . page 76

CGAL::Gmpfi

Class

Definition

An object of the class *Gmpfi* is a closed interval, with endpoints represented as *Gmpfr* floating-point numbers. An interval can have finite or infinite endpoints and its meaning is straightforward. It can also have one (or both) *NaN* endpoint(s): this indicates that an invalid operation has been performed and that the resulting interval has no mathematical meaning.

All the operations of *Gmpfi* were designed in such a way that the mathematical correct result is always contained in the resulting interval.

This type is *ImplicitInteroperable* with *Gmpfr*, *Gmpz*, *Gmpq*, `long`, `unsigned long`, `int`, `double` and `long double`.

#include <CGAL/Gmpfi.h>

Is Model for the Concepts

FieldWithKthRoot
RealEmbeddable

Types

Gmpfi:: Precision_type — Type representing the precision (number of bits used to represent the mantissa) of a number.

Creation

All the constructors accept an optional last argument: a precision (a *Precision_type*, which can be used to specify the precision of the *Gmpfr* endpoints. If none is specified, the default precision will be used. As the endpoints are represented with a fixed number of bits, they may need to be rounded. In this case, the number from which the *Gmpfi* was constructed is guaranteed to be included in the constructed interval.

Gmpfi i(Precision_type p=get_default_precision());

creates an uninitialized *Gmpfi* interval *i*.

template <class T>
Gmpfi i(T t, Precision_type p=get_default_precision());

creates a *Gmpfi* initialized with the value of *t*. *T* is *Gmpfr*, *Gmpq*, or any type from which *Gmpfr* can be constructed from. The rounding of the endpoints will guarantee that *t* is included in *i*.

Gmpfi i(Gmpfr left, Gmpfr right, Precision_type p=get_default_precision());

creates a *Gmpfi* initialized with endpoints *left* and *right*. The rounding of the endpoints will guarantee that $[left, right]$ is included in *i*.

template<class L, class R>
Gmpfi i(std::pair<L,R> endpoints, Precision_type p=get_default_precision());

creates a *Gmpfi* initialized with endpoints *endpoints.first* and *endpoints.second*. *L* and *R* are types from which *Gmpfr* can be constructed from. The rounding of the endpoints will guarantee that [*endpoints.first*, *endpoints.second*] is included in *i*.

Operations

Gmpfr *i.inf() const* Returns the smallest (or *inferior*) *Gmpfr* endpoint of *i*.

Gmpfr *i.sup() const* Returns the largest (or *superior*) *Gmpfr* endpoint of *i*.

static Precision_type *i.get_default_precision()*

Returns the default precision.

static Precision_type *i.set_default_precision(Precision_type prec)*

Sets the default precision to *prec* and returns the old value.

Precision_type *i.get_precision() const*

Returns the precision of *i*.

Gmpfi *i.round(Precision_type p) const*

Returns the value of the number, rounded with precision *p*.

Arithmetic operators `+`, `-`, `*` and `/` are overloaded, but special care must be taken when applying them. The precision of an operation between two *Gmpfi*s is defined as the maximum of the operands precision and the default precision.

The second operand of the former operations can be a `Gmpfi`, `Gmpfr`, `int`, `long`, `unsigned`, `unsigned long`, `Gmpz` or `Gmpq`. The precision of an operation between a *Gmpfi* and a number of another type is defined as the *Gmpfi*'s precision (even when operating with a `Gmpfr`).

To specify the rounding mode and/or the precision to perform an operation, this class provides the four static functions `add`, `sub`, `mul` and `div`. Only one of them is shown here, since their interfaces are similar:

static Gmpfi *add(a, b, Precision_type p=0)*

When the precision is not specified in this family of functions, it is defined as in the overloaded operators.

Other arithmetic functions provided by the class are:

Gmpfi *i.abs(Precision_type p) const*

Returns the absolute value of i, with precision p. If p is not specified, the precision used is the maximum between i's precision and the default.

Gmpfi *i.sqrt(Precision_type p) const*

Returns the square root of i, with precision p. If p is not specified, the precision used is the maximum between i's precision and the default.

Gmpfi *i.kthroot(int k, Precision_type p) const*

Returns the k-th root of i, with precision p. If p is not specified, the precision used is the maximum between i's precision and the default.

Gmpfi *i.square(Precision_type p) const*

Returns the square of i, with precision p. If p is not specified, the precision used is the maximum between i's precision and the default.

std::pair<double,double>

i.to_interval() const Returns an interval of doubles which contains i. If a rounded endpoint does not fit in a double, sets its value to plus or minus infinity and the *overflow* or *underflow* flag.

std::pair<double,long>

i.to_double_exp() const

Returns (m,e) such that $m \times 2^e$ is the center of i, rounded to nearest. If one of the endpoints of i is *NaN* or infinity, then the corresponding double is returned, leaving the exponent undefined and setting the appropriate error flag.

std::pair<std::pair<double,double>,long>

i.to_interval_exp() const

Returns $((m_1,m_2),e)$, such that $[m_1 \times 2^e, m_2 \times 2^e]$ contains i. If one of the endpoints of i is *NaN* or infinity, then the corresponding doubles are returned, leaving the exponent undefined and setting the appropriate error flag.

Comparisons

The semantics of the comparison operators is the same than on *Interval_nt<Protected>*. The result of the comparison is always an *Uncertain<bool>* (this type is convertible to *bool*, but may throw an exception). If compared intervals have no common points, the result is *true* or *false*; otherwise, *Uncertain<bool>::indeterminate()* will be returned.

In the same way, we can explain the semantics of *Uncertain<Comparison_result>* and *Uncertain<Sign>*.

With the semantics described above, this class provides comparisons between *Gmpfi* and *Gmpfi*, *Gmpfr*, *long*, *unsigned long*, *int*, *double*, *Gmpz* and *Gmpq*. Comparison operators ==, !=, >, <, >= and <= are overloaded.

The class provides also functions to test efficiently some special kinds of comparisons:

bool	*i.is_same(j) const*	Returns *true* iff left endpoints of *i* and *j* are equal and right endpoints of them are also equal. Note that this does not mean equality between *i* and *j*.
bool	*i.do_overlap(j) const*	Returns *true* iff *i* and *j* overlap, i.e., iff they have points in common.
Uncertain<Comparison_result>		
	i.compare(j) const	If *i* and *j* do not overlap, this function returns the result of the comparison. Otherwise, it returns *indeterminate*.

Query Functions

bool	*i.is_point() const*	Returns *true* iff both endpoints are equal.
bool	*i.is_nan() const*	Returns *true* iff at least one of the endpoints is *NaN*.
bool	*i.is_inf() const*	Returns *true* iff at least one of the endpoints is plus or minus infinity.
bool	*i.is_number() const*	Returns *true* iff *i* is a bounded interval, i.e. its endpoints are neither invalid nor infinite.
Uncertain<bool>	*i.is_zero() const*	Returns *true* if both endpoints are zero, *false* if the interval does not contain zero and *indeterminate* otherwise.
Uncertain<bool>	*i.is_one() const*	Returns *true* if both endpoints are one, *false* if the interval does not contain one and *indeterminate* otherwise.
Uncertain<Sign>	*i.sign() const*	If all numbers contained in the interval have the same sign, this function returns it. Otherwise it returns *indeterminate*.
Uncertain<bool>	*i.is_positive() const*	Returns *true* if all numbers contained in the interval are positive, false if all of them are negative or zero and *indeterminate* otherwise.
Uncertain<bool>	*i.is_negative() const*	Returns *true* if all numbers contained in the interval are negative, false if all of them are positive or zero and *indeterminate* otherwise.

Input/Output

std::istream&	*std::istream &is >> i*	Reads *i* from *is*. *is* must have the form *[inf,sup]*, where *inf* and *sup* have valid *Gmpfr* input formats.
std::ostream&	*std::ostream &os << i*	Writes *i* to *os*, in the form *[i.inf(),i.sup()]*. The endpoints are written according to the *Gmpfr* formatting.

Implementation

All interval operations are performed by the MPFI library. The class *Gmpfi* is not reference counted, but its members are.

See Also

Gmpfr . . . . page **??**
Interval_nt<Protected> . . . . page **??**
Uncertain<T> . . . . page **??**
RealEmbeddable . . . . page 123
FieldWithKthRoot . . . . page 76

Class

CGAL::MP_Float

Definition

An object of the class *MP_Float* is able to represent a floating point value with arbitrary precision. This number type has the property that additions, subtractions and multiplications are computed exactly, as well as the construction from *float*, *double* and *long double*.

Division and square root are not enabled by default since CGAL release 3.2, since they are computed approximately. We suggest that you use rationals like *Quotient<MP_Float>* when you need exact divisions.

Note on the implementation : although the mantissa length is basically only limited by the available memory, the exponent is currently represented by a (integral valued) *double*, which can overflow in some circumstances. We plan to also have a multiprecision exponent to fix this issue.

#include <CGAL/MP_Float.h>

Is Model for the Concepts

EuclideanRing.

RealEmbeddable

Creation

MP_Float m;	introduces an uninitialized variable *m*.
MP_Float m(MP_Float);	copy constructor.
MP_Float m(int i);	introduces the integral value i.
MP_Float m(float d);	introduces the floating point value d (exact conversion).
MP_Float m(double d);	introduces the floating point value d (exact conversion).
MP_Float m(long double d);	introduces the floating point value d (exact conversion).

Operations

std::ostream& *std::ostream& out << m*

writes a double approximation of *m* to the ostream *out*.

std::istream& *std::istream& in >> & m*

reads a *double* from *in*, then converts it to an *MP_Float*.

MP_Float *approximate_division(a, b)*

computes an approximation of the division by converting the operands to *double*, performing the division on *double*, and converting back to *MP_Float*.

MP_Float	*approximate_sqrt(a)*	computes an approximation of the square root by converting the operand to *double*, performing the square root on *double*, and converting back to *MP_Float*.

Implementation

The implementation of *MP_Float* is simple but provides a quadratic complexity for multiplications. This can be a problem for large operands. For faster implementations of the same functionality with large integral values, you may want to consider using *GMP* or *LEDA* instead.

Class

CGAL::Interval_nt<Protected>

Definition

This section describes briefly what interval arithmetic is, its implementation in CGAL, and its possible use by geometric programs. The main reason for having interval arithmetic in CGAL is its integration into the filtered robust and fast predicates scheme, but we also provide a number type so that you can use it separately if you find any use for it, such as interval analysis, or to represent data with tolerance...

The purpose of interval arithmetic is to provide an efficient way to bound the roundoff errors made by floating point computations. You can choose the behavior of your program depending on these errors. You can find more theoretical information on this topic in [BBP01].

Interval arithmetic is a large concept and we will only consider here a simple arithmetic based on intervals whose bounds are *double*s. So each variable is an interval representing any value inside the interval. All arithmetic operations (+, -, $*$, $/$, $\sqrt{}$, *square()*, *min()*, *max()* and *abs()*) on intervals preserve the inclusion. This property can be expressed by the following formula (x and y are real, X and Y are intervals, $\mathcal{OP}$ is an arithmetic operation):

$$\forall\, x \in X, \forall\, y \in Y, (x\ \mathcal{OP}\ y) \in (X\ \mathcal{OP}\ Y)$$

For example, if the final result of a sequence of arithmetic operations is an interval that does not contain zero, then you can safely determine its sign.

#include <CGAL/Interval_nt.h>

Parameters

The template parameter *Protected* is a Boolean parameter, which defaults to *true*. It provides a way to select faster computations by avoiding rounding mode switches, at the expense of more care to be taken by the user (see below). The default value, *true*, is the safe way, and takes care of proper rounding mode changes. When specifying *false*, the user has to take care about setting the rounding mode towards plus infinity before doing any computations with the interval class. He can do so using the *Protect_FPU_rounding* class for example.

Is Model for the Concepts

FieldWithSqrt

RealEmbeddable

Types

The class *Interval_nt* defines the following types: *typedef double* *value_type;* The type of the bounds of the in

typedef Uncertain_conversion_exception

unsafe_comparison; The type of the exceptions raised when uncertain comparisons are performed.

Interval_nt<Protected>:: Protector — A type whose default constructor and destructor allow to protect a block of code from FPU rounding modes necessary for the computations with *Interval_nt<false>*. It does nothing for *Interval_nt<true>*. It is implemented as *Protect_FPU_rounding<!Protected>*.

Creation

Interval_nt<Protected> I(int i); — introduces the interval [i;i].

Interval_nt<Protected> I(double d); — introduces the interval [d;d].

Interval_nt<Protected> I(double i, double s); — introduces the interval [i;s].

Interval_nt<Protected> I(std::pair<double, double> p);

introduces the interval [*p.first*;*p.second*].

Operations

All functions required by a class to be considered as a CGAL number type (see 5) are present, as well as the utility functions, sometimes with a particular semantic which is described below. There are also a few additional functions.

Interval_nt	*I / Interval_nt J*	returns $[-\infty;+\infty]$ when the denominator contains 0.
Interval_nt	*sqrt(Interval_nt I)*	returns $[0;\sqrt{upper\_bound(I)}]$ when only the lower bound is negative (expectable case with roundoff errors), and is unspecified when the upper bound also is negative (unexpected case).
double	*to_double(Interval_nt I)*	returns the middle of the interval, as a double approximation of the interval.
double	*I.inf()*	returns the lower bound of the interval.
double	*I.sup()*	returns the upper bound of the interval.
bool	*I.is_point()*	returns whether both bounds are equal.
bool	*I.is_same(Interval_nt J)*	returns whether both intervals have the same bounds.
bool	*I.do_overlap(Interval_nt J)*	returns whether both intervals have a non empty intersection.

The comparison operators ($<$, $>$, $<=$, $>=$, $==$, $!=$, *sign()* and *compare()*) have the following semantic: it is the intuitive one when for all couples of values in both intervals, the comparison is identical (case of non-overlapping intervals). This can be expressed by the following formula (x and y are real, X and Y are intervals, $\mathcal{OP}$ is a comparison operator):

$$(\forall x \in X, \forall y \in Y, (x\ \mathcal{OP}\ y) = true) \Rightarrow (X\ \mathcal{OP}\ Y) = true$$

and

$$(\forall x \in X, \forall y \in Y, (x\ \mathcal{OP}\ y) = false) \Rightarrow (X\ \mathcal{OP}\ Y) = false$$

Otherwise, the comparison is not safe, and we specify this by returning a type encoding this uncertainty, namely using *Uncertain<bool>* or *Uncertain<Sign>*, which can be probed for uncertainty explicitly, and which has a conversion to the normal type (e.g. *bool*) which throws an exception when the conversion is not certain. Note that each failed conversion increments a profiling counter (see *CGAL_PROFILE*), and then throws the exception of type *unsafe_comparison*.

Uncertain<bool>	*Interval_nt i* $<$ *Interval_nt j*
Uncertain<bool>	*Interval_nt i* $>$ *Interval_nt j*
Uncertain<bool>	*Interval_nt i* $<=$ *Interval_nt j*
Uncertain<bool>	*Interval_nt i* $>=$ *Interval_nt j*
Uncertain<bool>	*Interval_nt i* $==$ *Interval_nt j*
Uncertain<bool>	*Interval_nt i* $!=$ *Interval_nt j*
Uncertain<Comparison_result>	*compare(Interval_nt i, Interval_nt j)*
Uncertain<Sign>	*sign(Interval_nt i)*

typedef Interval_nt<false>

Interval_nt_advanced;

This typedef (at namespace CGAL scope) exists for backward compatibility, as well as removing the need to remember the Boolean value for the template parameter.

advanced

Implementation

The operations on *Interval_nt* with the default parameter *true*, are automatically protected against rounding modes, and are thus slower than those on *Interval_nt_advanced*, but easier to use. Users that need performance are encouraged to use *Interval_nt_advanced* instead.

Changing the rounding mode affects all floating point computations, and might cause problems with parts of your code, or external libraries (even CGAL), that expect the rounding mode to be the default (round to the nearest).

We provide two interfaces to change the rounding mode. The first one is to use a protector object whose default constructor and destructor will take care of changing the rounding mode. The protector is implemented using *Protect_FPU_rounding*.

The second one is the following detailed set of functions :

typedef int	*FPU_CW_t;*	The type used by the following functions to deal with rounding modes. This is usually an *int*.
void	*FPU_set_cw(FPU_CW_t R)*	
		sets the rounding mode to *R*.
FPU_CW_t	*FPU_get_cw(void)*	returns the current rounding mode.
FPU_CW_t	*FPU_get_and_set_cw(FPU_CW_t R)*	
		sets the rounding mode to *R* and returns the old one.

The macros *CGAL_FE_TONEAREST*, *CGAL_FE_TOWARDZERO*, *CGAL_FE_UPWARD* and *CGAL_FE_DOWNWARD* are the values corresponding to the rounding modes.

Example

Protecting an area of code that uses operations on the class *Interval_nt_advanced* can be done in the following way:

```
{
  Interval_nt_advanced::Protector P;
  ... // The code to be protected.
}
```

The basic idea is to use the directed rounding modes specified by the *IEEE 754* standard, which are implemented by almost all processors nowadays. It states that you have the possibility, concerning the basic floating point operations $(+,-,*,/,\sqrt{})$ to specify the rounding mode of each operation instead of using the default, which is set to 'round to the nearest'. This feature allows us to compute easily on intervals. For example, to add the two intervals [a.i;a.s] and [b.i;b.s], compute $c.i = a.i + b.i$ rounded towards minus infinity, and $c.s = a.s + b.s$ rounded towards plus infinity, and the result is the interval [c.i;c.s]. This method can be extended easily to the other operations.

The problem is that we have to change the rounding mode very often, and the functions of the C library doing this operation are slow and not portable. That's why assembly versions are used as often as possible. Another trick is to store the opposite of the lower bound, instead of the lower bound itself, which allows us to never change the rounding mode inside simple operations. Therefore, all basic operations, which are in the class *Interval_nt_advanced* assume that the rounding mode is set to 'round to infinity', and everything works with this correctly set.

So, if the user needs the speed of *Interval_nt_advanced*, he must take care of setting the rounding mode to 'round to infinity' before each block of operations on this number type. And if other operations might be affected by this, he must take care to reset it to 'round to the nearest' before they are executed.

Notes:

- On Intel platforms (with any operating system and compiler), due to a misfeature of the floating point unit, which does not handle exactly IEEE compliant operations on doubles, we are forced to use a workaround which slows down the code, but is only useful when the intervals can overflow or underflow. If you know that the intervals will never overflow nor underflow for your code, then you can disable this workaround with the flag *CGAL_IA_NO_X86_OVER_UNDER_FLOW_PROTECT*. Other platforms are not affected by this flag.

- When optimizing, compilers usually propagate the value of variables when they know it's a constant. This can break the interval routines because the compiler then does some floating point operations on these constants with the default rounding mode, which is wrong. This kind of problem is avoided by stopping constant propagation in the interval routines. However, this solution slows down the code and is rarely useful, so you can disable it by setting the flag *CGAL_IA_DONT_STOP_CONSTANT_PROPAGATION*.

advanced

CGAL::Lazy_exact_nt<NT>

Class

Definition

An object of the class *Lazy_exact_nt<NT>* is able to represent any real embeddable number which *NT* is able to represent. The idea is that *Lazy_exact_nt<NT>* works exactly like *NT*, except that it is expected to be faster because it tries to only compute an approximation of the value, and only refers to *NT* when needed. The goal is to speed up exact computations done by any exact but slow number type *NT*.

NT must be a model of concept *RealEmbeddable*.
NT must be at least model of concept *IntegralDomainWithoutDivision*.

Note that some filtering mechanism is available at the predicate level using *Filtered_predicate* and *Filtered_kernel*.

#include <CGAL/Lazy_exact_nt.h>

Is Model for the Concepts

IntegralDomainWithoutDivision	same as *NT*
RealEmbeddable	
Fraction	if *NT* is a *Fraction*

Creation

Lazy_exact_nt<NT> m;	introduces an uninitialized variable *m*.
Lazy_exact_nt<NT> m(int i);	introduces the integral value *i*.
Lazy_exact_nt<NT> m(double d);	introduces the floating point value *d* (works only if *NT* has a constructor from a double too).
Lazy_exact_nt<NT> m(NT n);	introduces the value *n*.
template <class NT1> *Lazy_exact_nt<NT> m(Lazy_exact_nt<NT1> n);*	introduces the value *n*. *NT1* needs to be convertible to *NT* (and this conversion will only be done if necessary).

Operations

NT	*m.exact()*	returns the corresponding NT value.
Interval_nt<true>	*m.approx()*	returns an interval containing the exact value.
Interval_nt<false>	*m.interval()*	returns an interval containing the exact value.

static void *m.set_relative_precision_of_to_double(double d)*

specifies the relative precision that *to_double()* has to fulfill. The default value is 10^{-5}.
Precondition: d¿0 and d¡1.

static double *m.get_relative_precision_of_to_double()*

returns the relative precision that *to_double()* currently fulfills.

std::ostream& *std::ostream& out << m*

writes *m* to ostream *out* in an interval format.

std::istream& *std::istream& in >> & m*

reads a *NT* from *in*, then converts it to a *Lazy_exact_nt<NT>*.

Example

```
#include <CGAL/Cartesian.h>
#include <CGAL/MP_Float.h>
#include <CGAL/Lazy_exact_nt.h>
#include <CGAL/Quotient.h>

typedef CGAL::Lazy_exact_nt<CGAL::Quotient<CGAL::MP_Float> > NT;
typedef CGAL::Cartesian<NT> K;
```

CGAL::Quotient<NT>

Class

Definition

An object of the class *Quotient<NT>* is an element of the field of quotients of the integral domain type *NT*. If *NT* behaves like an integer, *Quotient<NT>* behaves like a rational number. LEDA's class *rational* (see Section 5.5) has been the basis for *Quotient<NT>*. A *Quotient<NT> q* is represented as a pair of *NT*s, representing numerator and denominator.

NT must be at least model of concept *IntegralDomainWithoutDivision*.
NT must be a model of concept *RealEmbeddable*.

#include <CGAL/Quotient.h>

Is Model for the Concepts

Field
RealEmbeddable
Fraction

Creation

Quotient<NT> q;	introduces an uninitialized variable *q*.
template <class T> *Quotient<NT> q(T t);*	introduces the quotient *t/1*. NT needs to have a constructor from T.
template <class T> *Quotient<NT> q(Quotient<T> t);*	introduces the quotient *NT(t.numerator())/NT(t.denominator())*. NT needs to have a constructor from T.
Quotient<NT> q(NT n, NT d);	introduces the quotient *n/d*. *Precondition*: $d \neq 0$.

Operations

There are two access functions, namely to the numerator and the denominator of a quotient. Note that these values are not uniquely defined. It is guaranteed that *q.numerator()* and *q.denominator()* return values *nt_num* and *nt_den* such that *q* = *nt_num/nt_den*, only if *q.numerator()* and *q.denominator()* are called consecutively wrt *q*, i.e. *q* is not involved in any other operation between these calls.

NT	*q.numerator() const*	returns a numerator of *q*.
NT	*q.denominator() const*	returns a denominator of *q*.

The stream operations are available as well. They assume that corresponding stream operators for type *NT* exist.

std::ostream&	*std::ostream& out* << *q*	
		writes *q* to ostream *out* in format "`n/d`", where `n`==*q.numerator()* and `d`==*q.denominator()*.
std::istream&	*std::istream& in* >> & *q*	
		reads *q* from istream *in*. Expected format is "`n/d`", where `n` and `d` are of type *NT*. A single `n` which is not followed by a `/` is also accepted and interpreted as `n/1`.

The following functions are added to fulfill the CGAL requirements on number types.

double	*to_double(q)*	returns some double approximation to *q*.
bool	*is_valid(q)*	returns true, if numerator and denominator are valid.
bool	*is_finite(q)*	returns true, if numerator and denominator are finite.
Quotient<NT>	*sqrt(q)*	returns the square root of *q*. This is supported if and only if *NT* supports the square root as well.

CGAL::Number_type_checker<NT1,NT2,Comparator>

Definition

Number_type_checker is a number type whose instances store two numbers of types *NT1* and *NT2*. It forwards all arithmetic operations to them, and calls the binary predicate *Comparator* to check the equality of the instances after each modification, as well as for each comparison.

This is a debugging tool which is useful when dealing with number types.

Parameters

NT1 must be a model of some algebraic structure concept. *NT2* must be a model of the same algebraic structure concept. *NT1* and *NT2* must be *FromDoubleConstructible*. *Comparator* has to be a model of a binary predicate taking *NT1* as first argument, and *NT2* as second. The *Comparator* parameter has a default value which is a functor calling *operator==* between the two arguments.

#include <CGAL/Number_type_checker.h>

Is Model for the Concepts

IntegralDomainWithoutDivision (same as *NT1*)
RealEmbeddable **Creation**

Number_type_checker<NT1,NT2,Comparator> c;

introduces an uninitialized variable *c*.

Number_type_checker<NT1,NT2,Comparator> c(int i);

introduces the integral value i.

Number_type_checker<NT1,NT2,Comparator> c(double d);

introduces the floating point value d.

Number_type_checker<NT1,NT2,Comparator> c(NT1 n1, NT2 n2);

introduces a variable storing the pair *n1, n2*.

Operations

Some operations have a particular behavior documented here.

NT1	*c.n1() const*	returns a const reference to the object of type *NT1*.
NT2	*c.n2() const*	returns a const reference to the object of type *NT2*.
NT1&	*c.n1()*	returns a reference to the object of type *NT1*.
NT2&	*c.n2()*	returns a reference to the object of type *NT2*.

bool	*c.is_valid() const*	calls the *Comparator* binary predicate on the two stored objects and returns its result.
std::ostream&	*std::ostream& out << Number_type_checker c*	
		writes *c.n1()* to the ostream *out*.
std::istream&	*std::istream& in >> Number_type_checker& c*	
		reads an *NT1* from *in*, then converts it to an *NT2*, so a conversion from *NT1* to *NT2* is required here.

CGAL::Max<T,Less>

Functor
Class

Definition

The function object class *Max<T,Less>* returns the larger of two values. The order is induced by the second template argument `Less`. The default value for *Less* is *std::less*.

Note that *T* must be a model of *LessThanComparable* in case *std::less* is used.

#include <CGAL/utils_classes.h>

Is Model for the Concepts

AdaptableFunctor . . . . page **??**

Creation

Max<T,Less> max;	default constructor.
Max<T,Less> max(Less c);	The constructed object will use *c* to compare the arguments.

Operations

T	*operator()(T x, T y)*	returns the larger of *x* and *y*, with respect to the order induced by *Less*.

Functor
Class

CGAL::Min<T,Less>

Definition

The function object class *Min<T,Less>* returns the smaller of two values. The order is induced by the second template argument `Less`. The default value for *Less* is *std::less*.

Note that *T* must be a model of *LessThanComparable* in case *std::less* is used.

#include <CGAL/utils_classes.h>

Is Model for the Concepts

AdaptableFunctor . . . . page **??**

Creation

Min<T,Less> min;	default constructor.
Min<T,Less> min(Less c);	The constructed object will use *c* to compare the arguments.

Operations

T	*operator()(T x, T y)*	returns the larger of *x* and *y*, with respect to the order induced by *Less*.

CGAL::Is_valid<T>

Functor
Class

Definition

Not all values of a type need to be valid. The function object class *Is_valid<T>* checks this.

For example, an expression like *NT(0)/NT(0)* can result in an invalid number. Routines may have as a precondition that all values are valid.

#include <CGAL/utils_classes.h>

Is Model for the Concepts

AdaptableFunctor . . . . page **??**

bool	*operator()(T x)*	returns if the argument is valid.

Function

CGAL::max

Definition

The function *max* returns the larger of two values.

#include <CGAL/utils.h>

T	*max(T x, T y)*

Max . . . . page **??**

CGAL::min

Function

Definition

The function *min* returns the smaller of two values.

#include <CGAL/utils.h>

T	*min(T x, T y)*

See Also

Min . . . . page **??**

CGAL::is_valid

Definition

Not all values of a type need to be valid. The function *is_valid* returns whether the argument is valid.

#include <CGAL/utils.h>

bool *is_valid(T x)*

See Also

Is_valid . . . . page **??**

CGAL::Rational_traits<NT>

Class

Definition

The class *Rational_traits<NT>* can be used to determine the type of the numerator and denominator of a rational number type as *Quotient*, *Gmpq*, *mpq_class* or *leda_rational*.

#include <CGAL/Rational_traits.h>

Types

Rational_traits<NT>:: RT — the type of the numerator and denominator.

Operations

RT *t.numerator(NT r) const*

returns the numerator of r.

RT *t.denominator(NT r) const*

returns the denominator of r.

NT *t.make_rational(RT n, RT d) const*

constructs a rational number.

NT *t.make_rational(NT n, NT d) const*

constructs a rational number.

CGAL::simplest_rational_in_interval

Function

Definition

The function *simplest_rational_in_interval* computes the simplest rational number in an interval of two *double* values.

#include <CGAL/simplest_rational_in_interval.h>

Rational	*simplest_rational_in_interval(double d1, double d2)*	
		computes the rational number with the smallest denominator in the interval *[d1,d2]*.

Implementation

See Knuth, "Seminumerical algorithms", page 654, answer to exercise 4.53-39.

See Also

Rational CGAL::to_rational<Rational>(double d).

CGAL::to_rational

Function

Definition

The function *to_rational* computes the rational number representing a given double precision floating point number.

#include <CGAL/to_rational.h>

Rational	*to_rational(double d)*	computes the rational number that equals *d*.

Implementation

See Also

CGAL::simplest_rational_in_interval<Rational>(double d1, double d2).

Concept

RootOf_2

Definition

Concept to represent algebraic numbers of degree up to 2 over a *RealEmbeddable IntegralDomain* *RT*.

A model of this concept is associated to this *RT* via *CGAL::Root_of_traits*<*RT*>, which provides *Root_of_2* as a public type. Moreover, *CGAL::Root_of_traits*<*RT*> provides the public type *Root_of_1*, which is the quotient field of *RT*. We refer to *Root_of_1* as FT (for field type).

The model of *RootOf_2* is a *RealEmbeddable IntegralDomain*, which is *ImplicitInteroperable* with *RT*, *FT*. In particular, it provides the comparison operators ==, *!*=, <, >, <=, >= as well as the *sign* and *compare* functions needed to compare elements of types *RootOf_2, RT* and *FT*. It also provides all arithmetic operators +,-,*,/ among elements of type *RootOf_2* as well as mixed forms with *RT* and *FT*.

However, it is important to note that arithmetic operations among elements of *RootOf_2* are only allowed in the special case when they have been constructed from equations having the same discriminant, that is, if they are defined in the same algebraic extension of degree 2.

Besides construction from *int, RT* and *FT* the following functions provide special construction for extensions of degree 2:
- *make_root_of_2*
- *make_sqrt*

Refines

DefaultConstructible
CopyConstructible
FromIntConstructible
ImplicitInteroperable with RT
ImplicitInteroperable with FT

Operations

RootOf_2 &	$+= a$	*Precondition*: **this* and *a* are defined in the same extension.
RootOf_2	$a+b$	*Precondition*: *a* and *b* are defined in the same extension.

bool	$a == b$
bool	$a < b$

Same for operator `-`, `*`, `/`, `!=`, `<=`, `>`, `>=` as well as mixed forms with *RT* and *FT*.

Has Models

double (not exact)
CGAL::Sqrt_extension

See Also

CGAL::make_root_of_2<RT> ... page 217
CGAL::make_sqrt<RT> ... page 218
CGAL::compute_roots_of_2<RT,OutputIterator> ... page 219
CGAL::Root_of_traits<RT> ... page 216
AlgebraicKernelForCircles::PolynomialForCircles_2_2 ... page 1230
AlgebraicKernelForCircles ... page 1223

Class

CGAL::Root_of_traits<RT>

Definition

For a *RealEmbeddable IntegralDomain RT*, the class template *Root_of_traits<RT>* associates a type *Root_of_2*, which represents algebraic numbers of degree 2 over *RT*. Moreover, the class provides *Root_of_1*, which represents the quotient field of *RT*.

#include <CGAL/Root_of_traits.h>

Types

Root_of_traits<RT>:: Root_of_1	A *RealEmbeddable Field* representing the quotient field of *RT*.
Root_of_traits<RT>:: Root_of_2	Model of *RootOf_2*.

See Also

RootOf_2 . . . . page 214
CGAL::compute_roots_of_2<RT,OutputIterator> . . . . page 219
CGAL::make_root_of_2<RT> . . . . page 217

CGAL::make_root_of_2<RT>

Function

Definition

The function *make_root_of_2<RT>* constructs an algebraic number of degree 2 over a ring number type.

#include <CGAL/Root_of_traits.h>

template <typename RT>
Root_of_traits<RT>::Root_of_2

make_root_of_2(RT a, RT b, RT c, bool s)

Returns the smallest real root of the polynomial $aX^2 + bX + c$ if s is true, and the largest root is s is false.
Precondition: *RT* is an *IntegralDomainWithoutDivision*.
Precondition: The polynomial has at least one real root.

template <typename RT>
Root_of_traits<RT>::Root_of_2

make_root_of_2(RT alpha, RT beta, RT gamma)

Constructs the number $\alpha + \beta\sqrt{\gamma}$.
Precondition: *RT* is an *IntegralDomainWithoutDivision*.
Precondition: $\gamma \geq 0$

See Also

RootOf_2 . . . . page 214
CGAL::Root_of_traits<RT> . . . . page 216
CGAL::make_sqrt<RT> . . . . page 218
CGAL::compute_roots_of_2<RT,OutputIterator> . . . . page 219
CGAL::Sqrt_extension<NT,ROOT> . . . . page 220

Function

CGAL::make_sqrt<RT>

Definition

The function *make_sqrt<RT>* constructs a square root of a given value of type *RT*. Depending on the type *RT* the square root may be returned in a new type that can represent algebraic extensions of degree 2.

#include <CGAL/Root_of_traits.h>

template <typename RT>
Root_of_traits<RT>::Root_of_2

make_sqrt(RT x)	Returns $\sqrt{x}$. *Precondition*: *RT* is a *RealEmbeddable IntegralDomain*. *Precondition*: $x \leq 0$

See Also

RootOf_2 . . . . page 214
CGAL::make_root_of_2<RT> . . . . page 217
CGAL::Root_of_traits<RT> . . . . page 216

Function

CGAL::compute_roots_of_2<RT,OutputIterator>

Definition

The function *compute_roots_of_2<RT,OutputIterator>* solves a univariate polynomial as it is defined by the coefficients given to the function. The solutions are written into the given *OutputIterator*.

#include <CGAL/Root_of_traits.h>

template <typename RT, typename OutputIterator>
OutputIterator *compute_roots_of_2(RT a, RT b, RT c, OutputIterator oit)*

Writes the real roots of the polynomial $aX^2 + bX + c$ into *oit* in ascending order.
OutputIterator is required to accept *Root_of_traits<RT> ::Root_of_2*.
Multiplicities are not reported.

Precondition: *RT* is an *IntegralDomainWithoutDivision*.
Precondition: $a \neq 0$ or $b \neq 0$.

See Also

RootOf_2 .... page 214
CGAL::Root_of_traits<RT> .... page 216
CGAL::make_root_of_2<RT> .... page 217
CGAL::make_sqrt<RT> .... page 218
CGAL::Sqrt_extension<NT,ROOT> .... page 220

Class

CGAL::Sqrt_extension<NT,ROOT>

In addition, *Sqrt_extension* has another two default template arguments:
· *DifferentExtensionComparable = ::CGAL::Tag_false*
· *FilterPredicates = ::CGAL::Tag_false*
See below for more details.

Definition

An instance of this class represents an extension of the type *NT* by **one** square root of the type *ROOT*.
NT is required to be constructible from *ROOT*.
NT is required to be an *IntegralDomainWithoutDivision*.
Sqrt_extension<NT,ROOT> is *RealEmbeddable* if NT is *RealEmbeddable*.

For example, let *Integer* be some type representing $\mathbb{Z}$, then *Sqrt_extension<Integer,Integer>* is able to represent $\mathbb{Z}[\sqrt{root}]$ for some arbitrary Integer *root*.[2] The value of *root* is set at construction time, or set to zero if it is not specified.

Arithmetic operations among different extensions, say $\mathbb{Z}[\sqrt{a}]$ and $\mathbb{Z}[\sqrt{b}]$, are not supported. The result would be in $\mathbb{Z}[\sqrt{a},\sqrt{b}]$, which is not representable by *Sqrt_extension<Integer,Integer>*. **The user is responsible to check that arithmetic operations are carried out for elements from the same extensions only.** This is not tested by *Sqrt_extension<NT,ROOT>* for efficiency reasons. A violation of the precondition leads to undefined behavior. Be aware that for efficiency reasons the given *root* is stored as it is given to the constructor. In particular, an extension by a square root of a square is considered as an extension.

Since elements of *Sqrt_extension<NT,ROOT>* that lie in different extensions are not interoperable with respect to any arithmetic operations, the full value range of *Sqrt_extension<NT,ROOT>* does not represent an algebraic structure. However, each subset of the value range that represents the extension of NT by a particular square root is a valid algebraic structure, since this subset is closed under all provided arithmetic operations. From there, *Sqrt_extension<NT,ROOT>* can be used as if it were a model of an algebraic structure concept, with the following correspondence:

NT	*Sqrt_extension<NT,ROOT>*
IntegralDomainWithoutDivision	*IntegralDomainWithoutDivision*
IntegralDomain	*IntegralDomain*
UniqueFactorizationDomain	*IntegralDomain*
EuclideanRing	*IntegralDomain*
Field	*Field*

The extension of a *UniqueFactorizationDomain* or *EuclideanRing* is just an *IntegralDomain*, since the extension in general destroys the unique factorization property. For instance consider $\mathbb{Z}[\sqrt{10}]$, the extension of $\mathbb{Z}$ by $\sqrt{10}$: in $\mathbb{Z}[\sqrt{10}]$ the element 10 has two different factorizations $\sqrt{10}\cdot\sqrt{10}$ and $2\cdot 5$. In particular, the factorization is not unique.

If *NT* is a model of *RealEmbeddable* the type *Sqrt_extension* is also considered as *RealEmbeddable*. However, by default it is not allowed to compare values from different extensions for efficiency reasons. In case such a comparison becomes necessary, use the member function compare with the according Boolean flag. If such a comparison is a very frequent case, override the default of *DifferentExtensionComparable* by giving *::CGAL::Tag_true* as third template parameter. This effects the behavior of compare functions as well as the compare operators.

[2] $R[a]$ denotes the extension of a ring R by an element a. See also: http://mathworld.wolfram.com/ExtensionRing.html

The fourth template argument, *FilterPredicates*, triggers an internal filter that may speed up comparisons and sign computations. In case *FilterPredicates* is set to *CGAL::Tag_true* the type first computes a double interval containing the represented number and tries to perform the comparison or sign computation using this interval. Once computed, this interval is stored by the corresponding *Sqrt_extension* object for further usage. Note that this internal filter is switched off by default, since it may conflict with other filtering methods, such as *CGAL::Lazy_exact_nt<Sqrt_extension>*.

In case *NT* is not *RealEmbeddable*, *DifferentExtensionComparable* as well as *FilterPredicates* have no effect.

Is Model for the Concepts

Assignable	
CopyConstructible	
DefaultConstructible	
EqualityComparable	
ImplicitInteroperable	with int
ImplicitInteroperable	with NT
Fraction	if NT is a *Fraction*
RootOf_2	

Creation

Sqrt_extension<NT,ROOT>	*ext*;	Introduces an variable initialized with 0.
Sqrt_extension<NT,ROOT>	*ext(Sqrt_extension x);*	Copy constructor.
Sqrt_extension<NT,ROOT>	*ext(int i);*	Introduces an variable initialized with *i*.
Sqrt_extension<NT,ROOT>	*ext(NT x);*	Introduces an variable initialized with *x*.
Sqrt_extension<NT,ROOT>	*ext(int a0, int a1, int r);*	Constructor from int: *ext*$= a0 + a1 \cdot sqrt(r)$. *Precondition*: $r \neq 0$
Sqrt_extension<NT,ROOT>	*ext(NT a0, NT a1, ROOT r);*	General constructor: *ext*$= a0 + a1 \cdot sqrt(r)$. *Precondition*: $r \neq 0$

Operations

An object of type *Sqrt_extension* represent an expression of the form: $a0 + a1 * sqrt(root)$.

NT	*ext.a0() const*	Const access operator for a0
NT	*ext.a1() const*	Const access operator for a1
ROOT	*ext.root() const*	Const access operator for root
bool	*ext.is_extended() const*	Returns true in case root of *ext* is not zero. Note that $a1 == 0$ does not imply $root == 0$.
void	*ext.simplify()*	Simplifies the representation, in particular *root* is set to zero if $a1$ is zero, that is, *ext* becomes not extended. Moreover, it propagates the simplify command to members of *ext*. see also: *AlgebraicStructureTraits::Simplify*.
bool	*ext.is_zero() const*	returns true if *ext* represents the value zero.

CGAL::Sign	*ext.sign() const*	Determines the sign of *ext* by (repeated) squaring. *Precondition*: *Sqrt_extension* is *RealEmbeddable*.
Sqrt_extension	*ext.abs() const*	returns the absolute value of *ext*. *Precondition*: *Sqrt_extension* is *RealEmbeddable*.

CGAL::Comparison_result

ext.compare(Sqrt_extension y, bool in_same_extension = !DifferentExtensionComparable::value) const

Compares *ext* with y.
The optional bool *in_same_extension* indicates whether *ext* and *y* are in the same extension of NT.

Sqrt_extension *operator+(Sqrt_extension a, Sqrt_extension b)*

Precondition: *(a.root()==0 or b.root()==0 or a.root() == b.root())*

Sqrt_extension *operator-(Sqrt_extension a, Sqrt_extension b)*

Precondition: *(a.root()==0 or b.root()==0 or a.root() == b.root())*

Sqrt_extension *operator*(Sqrt_extension a, Sqrt_extension b)*

Precondition: *(a.root()==0 or b.root()==0 or a.root() == b.root())*

Sqrt_extension& *ext.operator+=(Sqrt_extension a)*

Precondition: *(this->root()==0 or a.root()==0 or this->root() == a.root())*

Sqrt_extension& *ext.operator-=(Sqrt_extension a)*

Precondition: *(this->root()==0 or a.root()==0 or this->root() == a.root())*

Sqrt_extension& *ext.operator*=(Sqrt_extension a)*

Precondition: *(this->root()==0 or a.root()==0 or this->root() == a.root())*

In case *NT* is only an *IntegralDomain* operator/ implements integral division. In case *NT* is a *Field* operator/ implements the field division.

Sqrt_extension *operator/(Sqrt_extension a, Sqrt_extension b)*

Precondition: *(a.root()==0 or b.root()==0 or a.root() == b.root())*

Sqrt_extension& *ext.operator/=(Sqrt_extension a)*

Precondition: *(this->root()==0 or a.root()==0 or this->root() == a.root())*

bool *operator==(Sqrt_extension a, Sqrt_extension b)*

Precondition: *(a.root()==0 or b.root()==0 or a.root() == b.root())*

bool *operator!=(Sqrt_extension a, Sqrt_extension b)*

Precondition: *(a.root()==0 or b.root()==0 or a.root() == b.root())*

In case *Sqrt_extension* is *RealEmbeddable*:

bool *operator<(Sqrt_extension a, Sqrt_extension b)*

Precondition: *(a.root()==0 or b.root()==0 or a.root() == b.root())*

bool *operator<=(Sqrt_extension a, Sqrt_extension b)*

Precondition: *(a.root()==0 or b.root()==0 or a.root() == b.root())*

bool *operator>(Sqrt_extension a, Sqrt_extension b)*

Precondition: *(a.root()==0 or b.root()==0 or a.root() == b.root())*

bool *operator>=(Sqrt_extension a, Sqrt_extension b)*

Precondition: *(a.root()==0 or b.root()==0 or a.root() == b.root())*

The stream operations are available as well. They assume that corresponding stream operators for type *NT* and *ROOT* exist.

std::ostream& *operator<<(std::ostream& os, ext)*

writes *ext* to ostream *os*. The format depends on the *CGAL::IO::MODE* of *os*.
In case the mode is *CGAL::IO::ASCII* the format is `EXT[a0,a1,root]`.
In case the mode is *CGAL::IO::PRETTY* the format is human readable.

std::istream& *operator>>(std::istream& is, ext)*

reads *ext* from istream *is* in format `EXT[a0,a1,root]`, the output format in mode *CGAL::IO::ASCII*

See Also

IntegralDomainWithoutDivision . . . . . page 69
IntegralDomain . . . . . page 71
Field . . . . . page 74
RealEmbeddable . . . . . page 123
ImplicitInteroperable . . . . . page 67
Fraction . . . . . page 142
RootOf_2 . . . . . page 214

CGAL::Tag_true . . . . . page 4232
CGAL::Tag_false . . . . . page 4233

Class

CGAL::Root_of_2<RT>

deprecated

This class is deprecated since CGAL-3.8. It is replaced by *Sqrt_extension*.

deprecated

CGAL::Sqrt_extension<NT,ROOT> . . . . page 220

CGAL::Set_ieee_double_precision

Definition

The IEEE754 standard specifies that the precision of double precision floating-point numbers should be 53 bits, with 11 bits for the exponent range.

Some processors violate this rule by providing excess precision during some computations (when values are in registers). This is the case of the x86 Intel processor and compatible processors (note that the SSE2 more recent alternative FPU is fortunately not affected by this issue). The effect of such excess precision can be a problem for some computations, since it can produce so-called double rounding effects, where actually *less* precision is actually provided! It can also be the root of non-deterministic computations depending on compiler optimizations or not (since this affects how long variables are kept in registers), for example numerical floating-point values get computed with slightly different results. Finally, it affects code that carefully makes use of cancellation properties, like *Residue*.

The class *Set_ieee_double_precision* provides a mechanism to set the correct 53 bits precision for a block of code. It does so by having a default constructor that sets a particular mode on the FPU which corrects the problem, and have its destructor reset the mode to its previous state.

Note that nothing can be done for the excess range of the exponent, which affects underflow and overflow cases, fortunately less frequent.

Note also that in the process of setting the correct precision, the rounding mode is also set to the nearest.

Moreover, some compilers provide a flag that performs this setting at the time of program startup. For example, GCC provides the option `-mpc64` since release 4.3 which does exactly this. Other compilers may have similar options.

Similarly, some third-party libraries may do the same thing as part of their startup process, and this is notably the case of LEDA (at least some versions of it). CGAL does not enforce this at startup as it would impact computations with long double performed by other codes in the same program.

Note that this property is notably required for proper functionning of the *Residue* class that performs modular arithmetic using efficient floating-point operations.

Note concerning Visual C++ 64-bit: due to a compiler bug, the stack unwinding process happenning when an exception is thrown does not correctly execute the restoring operation when the *Set_ieee_double_precision* object is destroyed. Therefore, for this configuration, some explicit code has to be added if you care about the state being restored.

#include <CGAL/FPU.h>

Creation

If the platform is not affected by the excess precision problem, this class becomes an empty class doing nothing.

Set_ieee_double_precision P;	Sets the precision of operations on double to 53bits. Note that the rounding mode is set to the nearest in the same process.

void *˜Set_ieee_double_precision()*

The precision and rounding modes are reset to the values they held before the constructor was called.

Also note that the following free function is also provided that does the same thing as the default constructor of *Set_ieee_double_precision* except that it does not perform the save and restore of the previous state.

void *force_ieee_double_precision()*

Sets the precision of operations on double to 53bits. Note that the rounding mode is set to the nearest in the same process.

See Also

CGAL::Protect_FPU_rounding<Protected> .... page 227
CGAL::Residue .... page 234

CGAL::Protect_FPU_rounding<Protected>

Class

Definition

Floating-point arithmetic, as specified by the IEEE-754 standard, allows to use so-called directed rounding for the following arithmetic operations: addition, subtraction, multiplication, division and square root. The default behavior is that the result of such an arithmetic operation is the closest floating-point number to the exact real result of the operation (rounding to the nearest). The other rounding modes are: round towards plus infinity, round towards minus infinity, and round towards zero.

Interval arithmetic uses such directed rounding modes to offer guaranteed enclosures for the evaluation of real functions, such as with CGAL's *Interval_nt* class.

In order to efficiently evaluate sequences of interval arithmetic operations, such as a geometric predicate computing for example a determinant, it is advised to reduce the number of rounding mode changes, which otherwise are performed for each arithmetic operation. CGAL exploits the fact that it is possible to compute a sequence of interval arithmetic operations by doing only one rounding mode change around the whole function evaluation in order to benefit from this optimization.

The class *Protect_FPU_rounding* allows to easily benefit from this. Its constructor saves the current rounding mode in the object, and then sets the current rounding mode to the value provided as argument to the constructor. The destructor sets the rounding mode back to the saved value. This allows to protect a block of code determined by a C++ scope, and have the destructor take care of restoring the value automatically.

The related class *Set_ieee_double_precision* allows to similarly protect a block of code from excess precision on some machines (x86 typically with the traditional FPU, not the more recent SSE2). Note that *Protect_FPU_rounding_mode*, when changing rounding modes, also sets the precision to the correct 64 bit precision, hence providing a similar effect to *Set_ieee_double_precision*. This notably affects the *Residue* class.

Note for Visual C++ 64-bit users: due to a compiler bug, the stack unwinding process happenning when an exception is thrown does not correctly execute the rounding mode restoration when the *Protect_FPU_rounding* object is destroyed. Therefore, for this configuration, some explicit code has to be added.

#include <CGAL/FPU.h>

Parameters

The template parameter *Protected* is a Boolean parameter, which defaults to *true*. It follows the same parameter of the *Interval_nt* class. When it is *false*, the constructor and the destructor of the class do nothing (this is meant to be used in a context where you know that the rounding mode change has been taken care of at a higher level in the call stack.

What follows describes the behavior when the parameter has its default value, *true*.

Creation

Protect_FPU_rounding<Protected> P(FPU_CW_t r = CGAL_FE_UPWARD);

The current rounding mode is saved in the object, and rounding mode is set to *r* which can be any of *CGAL_FE_TONEAREST*, *CGAL_FE_TOWARDZERO*, *CGAL_FE_UPWARD* (the default) and *CGAL_FE_DOWNWARD*.

void ~*Protect_FPU_rounding()*

The rounding mode is restored to the saved value.

See Also

CGAL::Set_ieee_double_precision . . . . page 225

Chapter 6

Modular Arithmetic

Michael Hemmer and Sylvain Pion

6.1 Introduction

Modular arithmetic is a fundamental tool in modern algebra systems. In conjunction with the Chinese remainder theorem it serves as the workhorse in several algorithms computing the gcd, resultant etc. Moreover, it can serve as a very efficient filter, since it is often possible to exclude that some value is zero by computing its modular correspondent with respect to one prime only.

First of all, this package introduces a type *CGAL::Residue*. It represents $\mathbb{Z}_{/p\mathbb{Z}}$ for some prime p. The prime number p is stored in a static member variable. The class provides static member functions to change this value. **Note that changing the prime invalidates already existing objects of this type.** However, already existing objects do not lose their value with respect to the old prime and can be reused after restoring the old prime. Since the type is based on double arithmetic the prime is restricted to values less than 2^{26}. The initial value of p is 67111067.

Please note that the implementation of class *CGAL::Residue* requires a mantissa precision according to the IEEE Standard for Floating-Point Arithmetic (IEEE 754). However, on some processors the traditional FPU uses an extended precision. Hence, it is indispensable that the proper mantissa length is enforced before performing any arithmetic operations. Moreover, it is required that numbers are rounded to the next nearest value. This can be ensured using *CGAL::Protect_FPU_rounding* with *CGAL_FE_TONEAREST*, which also enforces the required precision as a side effect.

advanced

In case the flag *CGAL_HAS_THREADS* is undefined the prime is just stored in a static member of the class, that is, *CGAL::Residue* is not thread-safe in this case. In case *CGAL_HAS_THREADS* the implementation of the class is thread safe using *boost::thread_specific_ptr*. However, this may cause some performance penalty. Hence, it may be advisable to configure CGAL with *CGAL_HAS_NO_THREADS*.

advanced

Moreover, the package introduces the concept *Modularizable*. An algebraic structure *T* is considered as *Modularizable* if there is a mapping from *T* into an algebraic structure that is based on the type *CGAL::Residue*. For scalar types, e.g. Integers, this mapping is just the canonical homomorphism into $\mathbb{Z}_{/p\mathbb{Z}}$ represented by *CGAL::Residue*. For compound types, e.g. Polynomials, the mapping is applied to the coefficients of the compound type. The mapping is provided by the class *CGAL::Modular_traits<T>*. The class *CGAL::Modular_* *traits<T>* is designed such that the concept *Modularizable* can be considered as optional, i.e., *CGAL::Modular_traits<T>* provides a tag that can be used for dispatching.

6.1.1 Example

In the following example modular arithmetic is used as a filter.

```
/* Modular arithmetic can be used as a filter, in this example modular
   arithmetic is used to avoid unnecessary gcd computations of
   polynomials.
   A gcd computation can be very costly due to coefficient growth within
   the
   Euclidean algorithm.

   The general idea is that firstly the gcd is computed with respect
   to one prime only. If this modular gcd is constant we can (in most
   cases)
   conclude that the actual gcd is constant as well.

   For this purpose the example introduces the function
   may_have_common_factor.
   Note that there are two versions of this function, namely for the case
   that the coefficient type is Modularizable and that it is not.
   If the type is not Modularizable the filter is just not applied and
   the
   function returns true.
*/

#include <CGAL/basic.h>

#ifdef CGAL_USE_GMP

#include <CGAL/Gmpz.h>
#include <CGAL/Polynomial.h>

// Function in case  Polynomial is Modularizable
template< typename Polynomial >
bool may_have_common_factor(
    const Polynomial& p1, const Polynomial& p2, CGAL::Tag_true){
  std::cout<< "The type is modularizable" << std::endl;

  // Enforce IEEE double precision and rounding mode to nearest
  // before useing modular arithmetic
  CGAL::Protect_FPU_rounding<true> pfr(CGAL_FE_TONEAREST);

  // Use Modular_traits to convert to polynomials with modular
    coefficients
  typedef CGAL::Modular_traits<Polynomial> MT;
  typedef typename MT::Residue_type MPolynomial;
  typedef typename MT::Modular_image Modular_image;
  MPolynomial mp1 = Modular_image()(p1);
  MPolynomial mp2 = Modular_image()(p2);
```

```
  // check for unlucky primes, the polynomials should not lose a degree
  typename CGAL::Polynomial_traits_d<Polynomial>::Degree  degree;
  typename CGAL::Polynomial_traits_d<MPolynomial>::Degree mdegree;
  if ( degree(p1) != mdegree(mp1)) return true;
  if ( degree(p2) != mdegree(mp2)) return true;

  // compute gcd for modular images
  MPolynomial mg  = CGAL::gcd(mp1,mp2);

  // if the modular gcd is not trivial: return true
  if ( mdegree(mg) > 0 ){
    std::cout << "The gcd may be non trivial" << std::endl;
    return true;
  }else{
    std::cout << "The gcd is trivial" << std::endl;
    return false;
  }
}

// This function returns true, since the filter is not applicable
template< typename Polynomial >
bool may_have_common_factor(
    const Polynomial&, const Polynomial&, CGAL::Tag_false){
  std::cout<< "The type is not modularizable" << std::endl;
  return true;
}

template< typename Polynomial >
Polynomial modular_filtered_gcd(const Polynomial& p1, const Polynomial&
   p2){
  typedef CGAL::Modular_traits<Polynomial> MT;
  typedef typename MT::Is_modularizable Is_modularizable;

  // Try to avoid actual gcd computation
  if (may_have_common_factor(p1,p2, Is_modularizable())){
    // Compute gcd, since the filter indicates a common factor
    return CGAL::gcd(p1,p2);
  }else{
    typename CGAL::Polynomial_traits_d<Polynomial>::Univariate_content
   content;
    typename CGAL::Polynomial_traits_d<Polynomial>::Construct_polynomial
   construct;
    return construct(CGAL::gcd(content(p1),content(p2))); // return
   trivial gcd
  }
}

int main(){
  CGAL::set_pretty_mode(std::cout);

  typedef CGAL::Gmpz NT;
  typedef CGAL::Polynomial<NT> Poly;
  CGAL::Polynomial_traits_d<Poly>::Construct_polynomial construct;
```

```
  Poly  f1=construct(NT(2), NT(6), NT(4));
  Poly  f2=construct(NT(12), NT(4), NT(8));
  Poly  f3=construct(NT(3), NT(4));

  std::cout << "f1         : " << f1 << std::endl;
  std::cout << "f2         : " << f2 << std::endl;

  std::cout << "compute modular filtered gcd(f1,f2): " << std::endl;
  Poly g1 = modular_filtered_gcd(f1,f2);
  std::cout << "gcd(f1,f2): " << g1 << std::endl;

  std::cout << std::endl;
  Poly p1 = f1*f3;
  Poly p2 = f2*f3;

  std::cout << "f3         : " << f3 << std::endl;
  std::cout << "p1=f1*f3   : " << p1 << std::endl;
  std::cout << "p2=f2*f3   : " << p2 << std::endl;

  std::cout << "compute modular filtered gcd(p1,p2): " << std::endl;
  Poly g2 = modular_filtered_gcd(p1,p2);
  std::cout << "gcd(p1,p2): " << g2 << std::endl;
}

#else

int main (){
  std::cout << " This examples needs GMP! " << std::endl;
}

#endif
```

File: examples/Modular_arithmetic/modular_filter.cpp

6.2 Design and Implementation History

The class *CGAL::Residue* is based on the C-code of Sylvain Pion et. al. as it was presented in [BEPP99].

The remaining part of the package is the result of the integration process of the NumeriX library of EXACUS [BEH+05] into CGAL.

Modular Arithmetic Reference Manual

Michael Hemmer

6.3 Classified Reference Pages

Concepts

Types

6.4 Alphabetical List of Reference Pages

Class

CGAL::Residue

Definition

The class *Residue* represents a finite field $\mathbb{Z}/p\mathbb{Z}$, for some prime number p.

The prime number p is stored in a static member variable. The class provides static member functions to change this value. **Note that changing the prime invalidates already existing objects of this type.** However, already existing objects do not lose their value with respect to the old prime and can be reused after restoring the old prime. Since the type is based on double arithmetic the prime is restricted to values less than 2^{26}. The initial value of p is 67111067.

Please note that the implementation of class *CGAL::Residue* requires a mantissa precision according to the IEEE Standard for Floating-Point Arithmetic (IEEE 754). However, on some processors the traditional FPU uses an extended precision. Hence, it is indispensable that the proper mantissa length is enforced before performing any arithmetic operations. Moreover, it is required that numbers are rounded to the next nearest value. This can be ensured using *CGAL::Protect_FPU_rounding* with *CGAL_FE_TONEAREST*, which also enforces the required precision as a side effect.

In case the flag *CGAL_HAS_THREADS* is undefined the prime is just stored in a static member of the class, that is, *Residue* is not thread-safe in this case. In case *CGAL_HAS_THREADS* the implementation of the class is thread safe using *boost::thread_specific_ptr*. However, this may cause some performance penalty. Hence, it may be advisable to configure *CGAL* with *CGAL_HAS_NO_THREADS*.

#include <CGAL/Residue.h>

Is Model for the Concepts

Field

Creation

Residue x;	introduces a variable x, which is initialized with zero;
Residue x(m);	copy constructor;
Residue x(int i);	introduces a variable x, which is initialized with $i\%p$;
Residue x(long i);	introduces a variable x, which is initialized with $i\%p$;

Operations

static int	*x.set_current_prime(int p)*	
		Static member function; sets current prime to the given value and returns the old prime.
static int	*x.get_current_prime()*	Static member function; returns the value of the current prime.

int	*x.get_value() const*	Returns the unique representative of $x$within the range $[-p/2, p/2]$, where p is the current prime.

Residue	$+a$
Residue	$-a$
Residue	$a+b$
Residue	$a-b$
Residue	$a*b$
Residue	a/b
Residue &	$x+=a$
Residue &	$x-=a$
Residue &	$x*=a$
Residue &	$x/=a$
Residue	$a==b$
Residue	$a\,!=b$

Modularizable

Definition

An algebraic structure is called Modularizable, if there is a suitable mapping into an algebraic structure which is based on the type *CGAL::Residue*. For scalar types, e.g. Integers, this mapping is just the canonical homomorphism into the type *CGAL::Residue* with respect to the current prime. For compound types, e.g. Polynomials, the mapping is applied to the coefficients of the compound type.

The mapping is provided via *CGAL::Modular_traits<Modularizable>*, being a model of *ModularTraits*.

Note that types representing rationals, or types which do have some notion of denominator, are not *Modularizable*. This is due to the fact that the denominator may be zero modulo the prime, which can not be represented.

Has Models

int . . . . . page 158
long . . . . . page **??**
CORE::BigInt . . . . . page 165
CGAL::Gmpz . . . . . page 175
leda::integer . . . . . page **??**
mpz_class . . . . . page 173

The following types are *Modularizable* iff their template arguments are. *CGAL::Lazy_exact_nt<NT>* . page 199
CGAL::Sqrt_extension<NT,ROOT> . . . . . page 220
CGAL::Polynomial<Coeff> . . . . . page 324

See Also

CGAL::Residue . . . . . page 234
CGAL::Modular_traits<T> . . . . . page 240

ModularTraits

Concept

Definition

A model of *ModularTraits* is associated to a specific *Type*. In case this associated type is *Modularizable*, this is indicated by the boolean tag *Is_modularizable*. The mapping into the *Residue_type* is provided by the functor *Modular_image*.

Types

A model of *ModularTraits* is supposed to provide:

ModularTraits:: Type	The associated type.
ModularTraits:: Is_modularizable	Tag indicating whether the associated type is modularizable. This is either *CGAL::Tag_true* or *CGAL::Tag_false*.
ModularTraits:: Residue_type	The type of the modular image. In case the type is not *Modularizable* this is undefined.

Functors

In case the associated type is *Modularizable* all functors are provided.
In case a functor is not provided, it is set to *CGAL::Null_functor*.

ModularTraits:: Modular_image	A model of *ModularTraits::ModularImage*
ModularTraits:: Modular_image_representative	A model of *ModularTraits::ModularImageRepresentative*

Has Models

CGAL::Modular_traits<T> . . . . page 240

See Also

CGAL::Residue . . . . page 234
Modularizable . . . . page 236

ModularTraits::ModularImage

Definition

This *AdaptableUnaryFunction* computes the modular image of the given value with respect to a homomorphism φ from the *ModularTraits::Type* into the *ModularTraits::Residue_type*.

The homomorphism preserves the mapping of *int* into both types , i.e., $\varphi(Type(i)) == Residue\_type(i)$.

Types

typedef ModularTraits::Residue_type

result_type;

typedef ModularTraits::Type

argument_type;

result_type	*fo(argument_type x)*	computes $\varphi(x)$.

Refines

AdaptableUnaryFunction

See Also

ModularTraits . . . . page 237

ModularTraits::ModularImageRepresentative

Definition

This *AdaptableUnaryFunction* returns a representative in the original type of a given modular image. More precisely, it implements the *rightinverse* of a proper restriction of the homomorphism φ, which is implemented by *ModularTraits::ModularImage*.

Types

typedef ModularTraits::Type

result_type;

typedef ModularTraits::Residue_type

argument_type;

result_type *fo(argument_type x)* computes $\varphi^{-1}(x)$.

Refines

AdaptableUnaryFunction

See Also

ModularTraits . . . . page 237

Class

CGAL::Modular_traits<T>

Definition

An instance of *Modular_traits<T>* is a model of *ModularTraits*, where T is the associated type.

#include <CGAL/Modular_traits.h>

Is Model for the Concepts

ModularTraits . . . . page 237

Chapter 7

Polynomial

Michael Hemmer

7.1 Fundamentals

Note that this is just a very brief introduction to polynomials. For a quick reference we refer to the Wikipedia or for a more elaborate introduction to any class book on elementary algebra.

A *polynomial* is either zero, or can be written as the sum of one or more non-zero *terms*. The number of terms is finite. A term consist of a constant *coefficient* and a *monomial*, that is, the product of zero or more variables. Each variable may have an exponent that is a non-negative integer. The exponent on a variable in a term is equal to the *degree* of that variable in that term. A term with no variables is called a constant term. The degree of a constant term is 0.

For example, $-7x^3y$ is a term. The coefficient is -7, the monomial is x^3y, comprised of the variables x and y, the degree of x is three, and the degree of y is one. The *total degree* of the entire term is the sum of the degrees in each variable. In the example above, the degree is $3+1=4$.

A one-variable (univariate) polynomial f of degree n has the following form:

$$f = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_2 x^2 + a_1 x + a_0$$

The coefficient a_0 is called the *constant coefficient*, a_n is called the *leading coefficient*. If f is not the zero polynomial the leading coefficient is not zero. The polynomial is called monic if $a_n = 1$. In case the coefficient domain of f possess a greatest common divisor (gcd) the *content* of f is the gcd of all coefficients of f. For instance, the content of $12x^3 + 6$ is 6.

A multivariate polynomial is a polynomial in more than one variable. According to the number of variables it is possible to further classify multivariate polynomials as bivariate, trivariate etc. In contrast to univariate polynomials the terms of a multivariate polynomial are not completely ordered by their total degree. However, given a certain order on the variables it is possible to define a lexicographic order on the terms. Given this order the leading coefficient of a multivariate polynomial is defined as the coefficient of the highest term. For instance the leading coefficient of the multivariate polynomial $p = 5x^3y + 7xy^2$ is 7, given that y has an higher order than x.

However, it is also possible to interpret a multivariate polynomial as a univariate polynomial in that variable.

For instance the trivariate polynomial

$$q = x^5 + 7x^2y^1z^2 + 13x^1y^2z^2 \in \mathbb{Z}[x,y,z]$$

may be interpreted as a univariate polynomial in z, that is, q is interpreted as an element of $R[z]$, with $R = \mathbb{Z}[x,y]$.

$$q = (13x^1y^2 + 7x^2y^1)z^2 + x^5z^0 \in R[z]$$

In this case the leading coefficient of q with respect to z is $13x^1y^2 + 7x^2y^1$ and x^5 becomes the 'constant' term.

A *homogeneous polynomial* is a polynomial whose terms do all have the same total degree. For example, $h = x^5 + 7x^2y^1z^2 + 13x^1y^2z^2$ is a homogeneous polynomial of degree 5, in three variables.

7.2 General Design

The package introduces a concept *Polynomial_d*, a concept for multivariate polynomials in d variables. Though the concept is written for an arbitrary number of variables, the number of variables is considered as fixed for a particular model of *Polynomial_d*. The concept also allows univariate polynomials.

First of all a model of *Polynomial_d* is considered as an algebraic structure, that is, the ring operations $\{+,-,\cdot\}$ are provided due to the fact that *Polynomial_d* refines at least the concept *IntegralDomainWithoutDivision*. However, a model of *Polynomial_d* has to be accompanied by a traits class *CGAL::Polynomial_traits_d< Polynomial_d>* being a model of *PolynomialTraits_d*. In principal the traits class provides all further functionalities on polynomials.

Given a d-variate polynomial over some base ring R there are at least two different possible views on such a polynomial.

- The recursive or univariate view: In this view, a polynomial is considered as an element of $R[x_0,\ldots,x_{d-2}][x_{d-1}]$. That is, the polynomial is treated as a univariate polynomial over the ring $R[x_0,\ldots,x_{d-2}]$.
- The symmetric or multivariate view: This view is almost symmetric with respect to all variables. It considers the polynomial as an element of $R[x_0,\ldots,x_{d-1}]$.

According to these two different views the traits class is required to provide two different coefficient types:

- *CGAL::Polynomial_traits_d<Polynomial_d>::Coefficient_type* representing $R[x_0,\ldots,x_{d-2}]$.
- *CGAL::Polynomial_traits_d<Polynomial_d>::Innermost_coefficient_type* representing the base ring R.

Another important type which is introduced by this package is *CGAL::Exponent_vector*. It is derived from *std::vector<int>* and used to identify multivariate monomials. For instance the exponent vector containing the sequence $[3,2,4]$ corresponds to the trivariate monomial $x_0^3x_1^2x_2^4$. Note that a vector with negative exponents is considered as invalid. However, we decided to in principal allow negative exponents as they may appear as intermediate results, in particular we did **not** derive from *std::vector<unsigned int>*.

7.3 Constructing a multivariate polynomial

First of all the concept *Polynomial_d* requires that the model is constructible from int. This is due to the fact that *Polynomial_d* refines *IntegralDomainWithoutDivision* which in turn refines *FromIntConstructible*. Of course this allows only the construction of constant polynomials.

In general a polynomial is constructed using the functor *CGAL::Polynomial_traits_d<Polynomial_d> ::Construct_polynomial* a model of *PolynomialTraits_d::ConstructPolynomial*. Basically there are two options:

- The polynomial is constructed from an iterator range with value type *CGAL::Polynomial_traits_d< Polynomial_d>::Coefficient_type*, where the *begin* iterator refers to the constant term (constant with respect to the outermost variable).
- The polynomial is constructed from an iterator range with value type *std::pair< CGAL::Exponent_vector, CGAL::Polynomial_traits_d<Polynomial_d>::Innermost_coefficient_type>*, where each pair defines the coefficient for the monomial defined by the exponent vector.

However, in some cases it might be more convenient to just construct the polynomials representing the different variables and to obtain the final polynomial using algebraic expressions. The most elegant way to construct a certain variable is *CGAL::Polynomial_traits_d<Polynomial_d>::Shift* being a model of *PolynomialTraits_d::Shift*.

7.3.1 Examples

The following example illustrates different ways to construct a bivariate polynomial:

```
#include <CGAL/config.h>
#include <CGAL/Polynomial.h>
#include <CGAL/Polynomial_traits_d.h>
#include <CGAL/Polynomial_type_generator.h>

int main(){
  CGAL::set_pretty_mode(std::cout);

  typedef CGAL::Polynomial_type_generator<int,2>::Type Poly_2;
  typedef CGAL::Polynomial_traits_d<Poly_2>            PT_2;
  typedef PT_2::Coefficient_type                       Poly_1;
  typedef PT_2::Innermost_coefficient_type             Integer;

  PT_2::Construct_polynomial construct_polynomial;

  // constructing a constant polynomial from int
  Poly_2 two(2); // = 2
  std::cout << "A constant polynomial: " << two << std::endl;


  // construction from an iterator range of univariate polynomials

  std::list<Poly_1> univariate_coeffs;
  univariate_coeffs.push_back(Poly_1(3));
  univariate_coeffs.push_back(Poly_1(0));
  univariate_coeffs.push_back(Poly_1(5));
  Poly_2 F = // 5*y^2 + 3

   construct_polynomial(univariate_coeffs.begin(),univariate_coeffs.end());
  std::cout << "The bivariate polynomial F: " << F << std::endl;
```

```
  // construction from an iterator range over monomials

  std::list<std::pair<CGAL::Exponent_vector, Integer> > innermost_coeffs;

   innermost_coeffs.push_back(std::make_pair(CGAL::Exponent_vector(0,0),-2));

   innermost_coeffs.push_back(std::make_pair(CGAL::Exponent_vector(3,5),2));
  Poly_2 G = // (2*x^3)*y^5 + (-2)

   construct_polynomial(innermost_coeffs.begin(),innermost_coeffs.end());
  std::cout << "The bivariate polynomial G: " << G << std::endl;

  //construction using shift
  PT_2::Shift shift;
  Poly_2 x = shift(Poly_2(1),1,0); // 'multiply' 1 by x_0^1
  Poly_2 y = shift(Poly_2(1),1,1); // 'multiply' 1 by x_1^1

  Poly_2 H = 5 * x * y + 3 * y * y; // = 3*y^2 + (5*x)*y
  std::cout << "The bivariate polynomial H: " << H << std::endl;
}
```

File: examples/Polynomial/construction.cpp

7.4 Coefficient Access

In order to obtain a certain coefficient the traits class provides several functors. Note that the functors do not allow a write access to the coefficients.

- *PolynomialTraits_d::GetCoefficient*: a model of this concept provides access to a coefficient in the univariate view, that is, it returns elements of $R[x_0,...,x_{d-2}]$.
- *PolynomialTraits_d::GetInnermostCoefficient*: a model of this concept provides access to a coefficient in the multivariate view, that is, it returns elements of R.
- *PolynomialTraits_d::LeadingCoefficient*: a model of this concept provides access to the leading coefficient in the univariate view.
- *PolynomialTraits_d::InnermostLeadingCoefficient*: a model of this concept provides access to the leading coefficient in the multivariate view, that is, it returns the (innermost) coefficient of the leading multivariate monomial. See also *PolynomialTraits_d::DegreeVector*.

7.4.1 Examples

The following example illustrates the application of the functors discussed above:

```
#include <CGAL/config.h>
#include <CGAL/Polynomial.h>
#include <CGAL/Polynomial_traits_d.h>
#include <CGAL/Polynomial_type_generator.h>
```

```
int main(){
  CGAL::set_pretty_mode(std::cout);
  typedef CGAL::Polynomial_type_generator<int,2>::Type Poly_2;
  typedef CGAL::Polynomial_traits_d<Poly_2>            PT_2;

  //construction using shift
  Poly_2 x = PT_2::Shift()(Poly_2(1),1,0); // = x^1
  Poly_2 y = PT_2::Shift()(Poly_2(1),1,1); // = y^1

  Poly_2 F // = (11*x^2 + 5*x)*y^4 + (7*x^2)*y^3
    = 11 * CGAL::ipower(y,4) * CGAL::ipower(x,2)
    + 5 * CGAL::ipower(y,4)  * CGAL::ipower(x,1)
    + 7 * CGAL::ipower(y,3)  * CGAL::ipower(x,2);
  std::cout << "The bivariate polynomial F: " << F <<"\n"<< std::endl;

  PT_2::Get_coefficient get_coefficient;
  std::cout << "Coefficient of y^0: "<< get_coefficient(F,0) <<
   std::endl;
  std::cout << "Coefficient of y^1: "<< get_coefficient(F,1) <<
   std::endl;
  std::cout << "Coefficient of y^2: "<< get_coefficient(F,2) <<
   std::endl;
  std::cout << "Coefficient of y^3: "<< get_coefficient(F,3) <<
   std::endl;
  std::cout << "Coefficient of y^4: "<< get_coefficient(F,4) <<
   std::endl;
  std::cout << "Coefficient of y^5: "<< get_coefficient(F,5) <<
   std::endl;
  std::cout << std::endl;

  PT_2::Leading_coefficient lcoeff;
  std::cout << "Leading coefficient with respect to y:           "
            << lcoeff(F)   // = 11*x^2 + 5*x
            << std::endl;

  PT_2::Get_innermost_coefficient get_icoeff;
  std::cout << "Innermost coefficient of monomial x^1y^4:        "
            << get_icoeff(F,CGAL::Exponent_vector(1,4)) // = 5
            << std::endl;

  PT_2::Innermost_leading_coefficient ilcoeff;
  std::cout << "Innermost leading coefficient with respect to y: "
            << ilcoeff(F) // = 11
            << std::endl;
}
```

File: examples/Polynomial/coefficient_access.cpp

7.5 Degree, total degree and degree vector

There are three functors in *PolynomialTraits_d* related to the degree of a polynomial.

- *PolynomialTraits_d::Degree*: a model of this concept returns the degree of the polynomial in the univariate view. By default this is the degree with respect to the outermost variable, but it is also possible to select another variable.
- *PolynomialTraits_d::TotalDegree*: a model of this concept returns the *total degree* of a polynomial. The polynomial is considered as a multivariate polynomial. The total degree is the maximum over the sums of the exponents of each multivariate monomial.
- *PolynomialTraits_d::DegreeVector*: a model of this concept returns the exponent vector of the leading monomial, where the monomial order is lexicographic and starts with the outermost variable. See also *PolynomialTraits_d::InnermostLeadingCoefficient*.

7.5.1 Examples

The following example illustrates the application of the functors discussed above:

```
#include <CGAL/config.h>
#include <CGAL/Polynomial.h>
#include <CGAL/Polynomial_traits_d.h>
#include <CGAL/Polynomial_type_generator.h>

int main(){
  CGAL::set_pretty_mode(std::cout);
  typedef CGAL::Polynomial_type_generator<int,2>::Type Poly_2;
  typedef CGAL::Polynomial_traits_d<Poly_2>            PT_2;

  //construction using shift
  Poly_2 x = PT_2::Shift()(Poly_2(1),1,0); // x_0^1
  Poly_2 y = PT_2::Shift()(Poly_2(1),1,1); // x_1^1

  Poly_2 F // = (11*x^2 + 5*x)*y^4 + (7*x^2)*y^3
    = 11 * CGAL::ipower(y,4) * CGAL::ipower(x,2)
    + 5 * CGAL::ipower(y,4)  * CGAL::ipower(x,1)
    + 7 * CGAL::ipower(y,3)  * CGAL::ipower(x,2);
  std::cout << "The bivariate polynomial F: " << F <<"\n"<< std::endl;

  PT_2::Degree degree;
  PT_2::Total_degree total_degree;
  PT_2::Degree_vector degree_vector;

  std::cout << "The degree of F with respect to y: "<< degree(F)
   // = 4
          << std::endl;
  std::cout << "The degree of F with respect to x: "<< degree(F,0)
   // = 2
          << std::endl;
  std::cout << "The total degree of F            : "<< total_degree(F)
   // = 6
          << std::endl;
  std::cout << "The degree vector of F           : "<<
   degree_vector(F)// = (2,4)
          << std::endl;
```

```
}
```

File: examples/Polynomial/degree.cpp

7.6 Changing the order of variables

Given for instance a bivariate polynomial it is conceivable that one wants to interchange the role of x and y. That is one wants to interpret the x as y and vice versa. For such a case the polynomial traits provides *PolynomialTraits_d::Swap*:
Given a polynomial p and to two indices i and j, the functor returns the polynomial in which x_i is substituted by x_j and vice versa, that is, the variables swap their positions. The order of the other variables remains untouched.

Another scenario is, that a particular variable should be moved to another position, for instance, it should become the outermost variable while the relative order of the other variables remains unchanged. For such a case the polynomial traits provides *PolynomialTraits_d::Move*.

Of course there is also a general method to interchange the order of variables, namely *PolynomialTraits_d::Permute*.

7.6.1 Examples

The following example illustrates the application of the functors discussed above:

```
#include <CGAL/config.h>
#include <CGAL/Polynomial.h>
#include <CGAL/Polynomial_traits_d.h>
#include <CGAL/Polynomial_type_generator.h>

int main(){
  CGAL::set_pretty_mode(std::cout);
  typedef CGAL::Polynomial_type_generator<int,3>::Type Poly_3;
  typedef CGAL::Polynomial_traits_d<Poly_3>            PT_3;

  //construction using shift
  Poly_3 x = PT_3::Shift()(Poly_3(1),1,0); // x_0^1
  Poly_3 y = PT_3::Shift()(Poly_3(1),1,1); // x_1^1
  Poly_3 z = PT_3::Shift()(Poly_3(1),1,2); // x_2^1


  Poly_3 F = x*y*y*z*z*z;
  std::cout << "The trivariate polynomial F: " << F << std::endl;
  std::cout << std::endl;

  PT_3::Swap swap;
  PT_3::Move move;
  PT_3::Permute permute;

  std::cout << "x and z swapped: "<< swap(F,0,2) // = x^3*y^2*z
```

```
                << std::endl;
  std::cout << "x and y swapped: "<< swap(F,0,1) // = x^2*y*z^3
            << std::endl << std::endl;

  std::cout << "x moved to outermost position            : "
            << move(F,0,2)                      // = x^2*y^3*z
            << std::endl;
  std::cout << "Same as swap(swap(F,0,1),1,2)           : "
            << swap(swap(F,0,1),1,2)            // = x^2*y^3*z
            << std::endl;

  std::cout << "Same as the permutation (0,1,2)->(2,0,1): ";
  std::vector<int> perm;
  perm.push_back(2);perm.push_back(0);perm.push_back(1);
  std::cout << permute(F,perm.begin(),perm.end())// = x^2*y^3*z
            << std::endl;

}
```

File: examples/Polynomial/swap_move.cpp

7.7 GCD and More

Since the concept *PolynomialTraits_d* refines the concept *AlgebraicStructureTraits* the polynomial traits provides functors for integral division, division with remainder, greatest common divisor, etc. But note that the algebraic structure of a polynomial depends on the algebraic structure of the innermost coefficient, for instance, a gcd is available if and only if the innermost coefficient is a *Field* or a *UniqueFactorizationDomain*. Hence, we can not provide a *gcd* if the innermost coefficient is just an *IntegralDomain* since it is simply not well defined[1]. However, if we would consider the polynomial over the quotient field of the integral domain the *gcd* would be well defined. The only problem is that the result can not be represented over the ring since it contains denominators. Therefore, the *PolynomialTraits_d* requires functors such as *PolynomialTraits_d::GcdUpToConstantFactor*. This functor computes the gcd of two polynomials up to a constant factor (utcf). That is, it returns the correct gcd for polynomials over the quotient field, but multiplied by some constant such that the result is representable with coefficients in the ring.

However, note that these 'utcf' functions are usually a bit faster than their strict counterparts. This is due to the fact that the 'utcf' functions are allowed to skip the computation of the correct constant factor. Note that in many cases the constant factor is in fact not needed. In particular if the polynomials are supposed to represent some zero set, that is, an algebraic curve or surface.

The concepts for the related functors are:

- *AlgebraicStructureTraits::Gcd*
  PolynomialTraits_d::GcdUpToConstantFactor
- *AlgebraicStructureTraits::IntegralDivision*
  PolynomialTraits_d::IntegralDivisionUpToConstantFactor

[1] An example for such a number type is the template *CGAL::Sqrt_extension<NT,ROOT>* representing an algebraic extension of degree two. This is just an *IntegralDomain* if NT is not a *Field*.

- *PolynomialTraits_d::UnivariateContent*
  PolynomialTraits_d::UnivariateContentUpToConstantFactor
- *PolynomialTraits_d::SquareFreeFactorize*
  PolynomialTraits_d::SquareFreeFactorizeUpToConstantFactor

Another analog functionality is the pseudo division. The related functors replace the usual division with remainder in case the Polynomial is not a *EuclideanRing*.
The concepts for the related functors are:

- *AlgebraicStructureTraits::Div_mod*
  PolynomialTraits_d::PseudoDivision
- *AlgebraicStructureTraits::Div*
  PolynomialTraits_d::PseudoDivisionQuotient
- *AlgebraicStructureTraits::Mod*
  PolynomialTraits_d::PseudoDivisionRemainder

7.7.1 Examples

The following example illustrates the application of some functors discussed above:

```
#include <CGAL/config.h>
#include <CGAL/Polynomial.h>
#include <CGAL/Polynomial_traits_d.h>
#include <CGAL/Polynomial_type_generator.h>

int main(){
  CGAL::set_pretty_mode(std::cout);
  typedef CGAL::Polynomial_type_generator<int,1>::Type Poly_1;
  typedef CGAL::Polynomial_traits_d<Poly_1>            PT_1;

  PT_1::Shift                         shift;
  PT_1::Gcd                           gcd;
  PT_1::Gcd_up_to_constant_factor gcd_utcf;
  PT_1::Multivariate_content          mcontent;
  PT_1::Canonicalize                  canonicalize;

  //construction using shift
  Poly_1 x = shift(Poly_1(1),1,0); // x^1

  // common factor 7 * (x^2-2)
  Poly_1 F = 21*(x-5)*(x*x-2); // = 21*x^3 + (-105)*x^2 + (-42)*x + 210
  Poly_1 G = 14*(x-3)*(x*x-2); // = 14*x^3 + (-42)*x^2 + (-28)*x + 84

  std::cout << "The univariate polynomial F: " << F << std::endl;
  std::cout << "The univariate polynomial G: " << G << std::endl;
  std::cout << "Common multivariate content:                  "
            << CGAL::gcd(mcontent(F),mcontent(G)) // = 7
            << std::endl;
  std::cout << "The gcd of F and G:                           "
            << gcd(F,G)                       // = 7*x^2 + (-14)
```

```
            << std::endl;
  std::cout << "The gcd up to constant factor of F and G: "
            << gcd_utcf(F,G)                    // = x^2 + (-2)
            << std::endl;
  std::cout << "Same as canonicalized gcd of F and G:     "
            << canonicalize(gcd_utcf(F,G))      // = x^2 + (-2)
            << std::endl;

}
```

File: examples/Polynomial/gcd_up_to_constant_factor.cpp

7.8 Evaluation and Substitution

Of course, it should also be possible to evaluate a polynomial or substitute its variables. We also require a special functor to determine whether a polynomial is zero at a given point. In case the inner most coefficient is *RealEmbeddable* the traits also must provide a function to compute the sign at a given point.
The concepts for the related functors are:

- *PolynomialTraits_d::Substitute*
- *PolynomialTraits_d::Evaluate*
- *PolynomialTraits_d::IsZeroAt*
- *PolynomialTraits_d::SignAt*

The traits is also required to provide variants of these functors that interpret the polynomial as a homogeneous polynomial by adding a virtual homogeneous variable such that each term has the same degree, namely the degree of the polynomial. Of course there is a difference between the univariate and multivariate view. For instance the polynomial

$$5x^3 + 7x - 3$$

has degree 3, hence it is interpreted as the homogeneous polynomial

$$5x^3 + 7xw^2 - 3w^3$$

by adding the homogeneous variable w. In case of the multivariate view each term is filled up by the homogeneous variable such that the degree of each term is equal to the total degree of the polynomial. Note that these functors may significantly improve efficiency. For instance, it is possible to determine the sign of a polynomial over integer coefficients at a rational point without changing the coefficient domain of the polynomial. For more details have a look at the following concepts:

- *PolynomialTraits_d::SubstituteHomogeneous*
- *PolynomialTraits_d::EvaluateHomogeneous*
- *PolynomialTraits_d::IsZeroAtHomogeneous*
- *PolynomialTraits_d::SignAtHomogeneous*

Note that substitute allows the substitution of the variables by any type that is *ExplicitInteroperable* with the innermost coefficient type. This is a very powerful tool since it allows the substitution of the variables by polynomials. However, for some standard manipulations such as translation or scaling we require special functors since they are expected to be faster than their equivalent implementation using substitution:

- *PolynomialTraits_d::Shift*

- *PolynomialTraits_d::Negate*
- *PolynomialTraits_d::Invert*
- *PolynomialTraits_d::Translate*
- *PolynomialTraits_d::TranslateHomogeneous*
- *PolynomialTraits_d::Scale*
- *PolynomialTraits_d::ScaleHomogeneous*

7.8.1 Examples

The following example illustrates the application of some functors discussed above:

```
#include <CGAL/config.h>
#include <CGAL/Polynomial.h>
#include <CGAL/Polynomial_traits_d.h>
#include <CGAL/Polynomial_type_generator.h>

int main(){
  CGAL::set_pretty_mode(std::cout);
  typedef CGAL::Polynomial_type_generator<int,2>::Type Poly_2;
  typedef CGAL::Polynomial_traits_d<Poly_2>            PT_2;

  //construction using shift
  Poly_2 x = PT_2::Shift()(Poly_2(1),1,0); // x^1
  Poly_2 y = PT_2::Shift()(Poly_2(1),1,1); // y^1


  Poly_2 F = 2*x*y + 3*CGAL::ipower(y,3);
  std::cout << "The bivariate polynomial F: " << F // = 3*y^3 + (2*x)*y
            << std::endl << std::endl;

  PT_2::Evaluate evaluate;
  PT_2::Evaluate_homogeneous hevaluate;

  // Evaluation considers a polynomials as univariate:
  std::cout << "F(5): " << evaluate(F,5)      // = 10*x + 375
            << std::endl;
  // Evaluate_homogeneous considers F as a homogeneous polynomial in
  // the outermost variable only, that is, F is interpreted as
  // F(u,v) = 2*x*u*v^2 + 3 * u^3
  std::cout << "F(5,7): " << hevaluate(F,5,7) // = 490*x + 375
            << std::endl << std::endl;


  PT_2::Substitute substitute;
  PT_2::Substitute_homogeneous hsubstitute;

  // Substitute considers a polynomials as multivariate, that is, the
  // new values for the variables are given by an iterator range
  // Note that the value type only has to be interoperable with the
   innermost
  // coefficient
  std::list<Poly_2> replacements;
  replacements.push_back(x-1); // replace x by x-1
  replacements.push_back(y);   // replace y by y, i.e., do nothing
```

```
  std::cout << "The bivariate polynomial F: " << F // = 3*y^3 + (2*x)*y
            << std::endl;
  std::cout << "F(x-1,y):   " // = 3*y^3 + (2*x + (-2))*y
            << substitute(F,replacements.begin(),replacements.end())
            << std::endl;
  // Substitute_homogeneous considers F as a homogeneous polynomial in
  // all variable, that is, F is interpreted as
  // F(x,y,w) = 2*x*y*w + 3 * y^3
  replacements.push_back(y);  // replace z by y

  std::cout << "F(x-1,y,y): " // = 3*y^3 + (2*x + (-2))*y^2
            << hsubstitute(F,replacements.begin(),replacements.end())
            << std::endl;
}
```

File: `examples/Polynomial/substitute.cpp`

7.9 Resultants, Subresultants and Sturm-Habicht sequences

The *PolynomialTraits_d* concept also provides more sophisticated functors for computations with polynomials – computing the resultant of two polynomials, their polynomial subresultant sequence, with or without cofactors, and their principal subresultant coefficients.

- PolynomialTraits_d::Resultant . . . . . . page 312
- PolynomialTraits_d::PolynomialSubresultants . . . . . . page 314
- PolynomialTraits_d::PolynomialSubresultantsWithCofactors . . . . . . page 316
- PolynomialTraits_d::PrincipalSubresultants . . . . . . page 318

Moreover, functors to compute the Sturm-Habicht sequence, with or without cofactors, and for the principal Sturm-Habicht coefficients exist.

- PolynomialTraits_d::SturmHabichtSequence . . . . . . page 319
- PolynomialTraits_d::SturmHabichtSequenceWithCofactors . . . . . . page 321
- PolynomialTraits_d::PrincipalSturmHabichtSequence . . . . . . page 323

For a formal definition of all used terms, we refer to the corresponding reference pages.

The principal Sturm-Habicht sequence allows to count the number of real roots of a polynomial using the function

- *CGAL::number_of_real_roots* . . . . . . page 377.

As input, this function requires an iterator range that represents the principal Sturm-Habicht coefficients. This might look complicated at a first sight, as one has to store the principal Sturm-Habicht sequence temporarily. However, we remark an important property of the (principal) Sturm-Habicht sequence. Having a polynomial $f_t(x)$ that depends on a parameter t, and its (principal) Sturm-Habicht coefficients $\mathrm{stha}_0(f_t), \ldots, \mathrm{stha}_n(f_t)$, evaluating $\mathrm{stha}_0(f_t)$ for $t = t_0$ yields a valid (principal) Sturm-Habicht sequence for f_{t_0}. The same holds for (principal) subresultants. Thus, it is enough in such situations to compute the sequence once for the parameter t, and call *CGAL::number_of_real_roots* for each specialized parameter value.

We finally remark that computing subresultants and Sturm-Habicht sequences introduces an enormous coefficient blow-up. An application of the functors therefore does not make sense for built-in integers except for toy examples. To avoid overflows, one should use arbitrary size integer types in real applications.

7.9.1 Examples

The following example illustrates how two compute resultants of two polynomials, and how to count the number of distinct real roots of a polynomial using its principal Sturm-Habicht coefficients.

```
#include <CGAL/config.h>
#include <CGAL/Polynomial.h>
#include <CGAL/Polynomial_traits_d.h>
#include <CGAL/Polynomial_type_generator.h>

#include <CGAL/Gmpz.h>

int main(){
  CGAL::set_pretty_mode(std::cout);

  typedef CGAL::Gmpz Int;

  typedef CGAL::Polynomial_type_generator<Int,1>::Type Poly_1;
  typedef CGAL::Polynomial_traits_d<Poly_1>            PT_1;

  //construction using shift
  Poly_1 x = PT_1::Shift()(Poly_1(1),1); // x^1

  Poly_1 F // = (x+1)^2*(x-1)*(2x-1)=2x^4+x^3-3x^2-x+1
    =   2 * CGAL::ipower(x,4) + 1 * CGAL::ipower(x,3)
      - 3 * CGAL::ipower(x,2) - 1 * CGAL::ipower(x,1)
      + 1 * CGAL::ipower(x,0);
  std::cout << "F=" << F << std::endl;

  Poly_1 G // = (x+1)*(x+3)=x^2+4*x+3
    =   1 * CGAL::ipower(x,2) + 4 * CGAL::ipower(x,1) + 3 *
   CGAL::ipower(x,0);
  std::cout << "G=" << G << std::endl;

  // Resultant computation:
  PT_1::Resultant resultant;

  std::cout << "The resultant of F and G is: " << resultant(F,G) <<
   std::endl;
  // It is zero, because F and G have a common factor

  // Real root counting:
  PT_1::Principal_sturm_habicht_sequence stha;
  std::vector<Int> psc;

  stha(F,std::back_inserter(psc));
```

```
  int roots = CGAL::number_of_real_roots(psc.begin(),psc.end());

  std::cout << "The number of real roots of F is: " << roots <<
    std::endl; // 3

  roots =  CGAL::number_of_real_roots(G);

  std::cout << "The number of real roots of G is: " << roots <<
    std::endl; // 2

  return 0;

}
```

File: examples/Polynomial/subresultants.cpp

7.10 Design and Implementation History

This package is the result of the integration process of the NumeriX library of EXACUS [BEH+05] into CGAL.

The class *CGAL::Polynomial<Coeff>* had been started by Michael Seel within CGAL as part of the Nef_2 package. As part of the EXACUS project it got significantly improved by Arno Eigenwillig and Michael Hemmer.

However, due to the recursive definition the class was rather restricted to the univariate view. Moreover, it is clear that depending on the context other classes that are symmetric in all variables or dedicated for sparse polynomials may be more efficient. As a consequence this package introduced the *CGAL::Polynomial_traits_d<Polynomial_d>* giving also the symmetric view on polynomials and the opportunity to introduce and use other classes representing polynomials within CGAL.

Polynomial
Reference Manual

Michael Hemmer

7.11 Classified Reference Pages

Polynomial_d

Concepts

Classes

Functions

7.12 Alphabetical List of Reference Pages

Polynomial_d

Definition

A model of Polynomial_d is representing a multivariate polynomial in $d \geq 1$ variables over some basic ring R. This type is denoted as the innermost coefficient. A model of Polynomial_d accompanied by a traits class *CGAL::Polynomial_traits_d<Polynomial_d>*, which is a model of *PolynomialTraits_d*. Please have a look at the concept *PolynomialTraits_d*, since nearly all functionality related to polynomials is provided by the traits.

Refines

IntegralDomainWithoutDivision

The algebraic structure of *Polynomial_d* depends on the algebraic structure of *Innermost_coefficient_type*:

Innermost_coefficient_type	*Polynomial_d*
IntegralDomainWithoutDivision	*IntegralDomainWithoutDivision*
IntegralDomain	*IntegralDomain*
UniqueFactorizationDomain	*UniqueFactorizationDomain*
EuclideanRing	*UniqueFactorizationDomain*
Field	*UniqueFactorizationDomain*

Note: In case the polynomial is univariate and the innermost coefficient is a *Field* the polynomial is model of *EuclideanRing*.

See Also

AlgebraicStructureTraits . . . . page 78
PolynomialTraits_d . . . . page 262

Has Models

CGAL::Polynomial<Coeff> . . . . page 324

Concept

PolynomialTraits_d

Definition

A model of *PolynomialTraits_d* is associated with a type *Polynomial_d*. The type *Polynomial_d* represents a multivariate polynomial. The number of variables is denoted as the dimension d of the polynomial, it is arbitrary but fixed for a certain model of this concept. Note that univariate polynomials are not excluded by this concept. In this case d is just set to one.

PolynomialTraits_d provides two different views on the multivariate polynomial.

- The recursive view: In this view, the polynomial is considered as an element of $R[x_0,\ldots,x_{d-2}][x_{d-1}]$. That is, the polynomial is treated as a univariate polynomial over the ring $R[x_0,\ldots,x_{d-2}]$.
- The symmetric or multivariate view: This view is symmetric with respect to all variables, considering the polynomial as an element of $R[x_0,\ldots,x_{d-1}]$.

Many functors consider the polynomial as a univariate polynomial in one variable. By default this is the outermost variable x_{d-1}. However, in general it is possible to select a certain variable.

Refines

AlgebraicStructureTraits

Constants

static const int *d;* The dimension and the number of variables respectively.

Types

PolynomialTraits_d:: Polynomial_d Type representing $R[x_0,\ldots,x_{d-1}]$.

PolynomialTraits_d:: Coefficient_type Type representing $R[x_0,\ldots,x_{d-2}]$.

PolynomialTraits_d:: Innermost_coefficient_type

Type representing the base ring R.

PolynomialTraits_d:: Coefficient_const_iterator

Const iterator used to iterate through all coefficients of the polynomial.

PolynomialTraits_d:: Innermost_coefficient_const_iterator

Const iterator used to iterate through all innermost coefficients of the polynomial.

PolynomialTraits_d:: template <typename T, int d> struct Rebind

This nested template class has to define a type *Other* which is a model of the concept *PolynomialTraits_d*, where *d* is the number of variables and *T* the *Innermost_coefficient_type*.

Functors

In case a functor is not provided it is set to *CGAL::Null_functor*.

PolynomialTraits_d:: Construct_polynomial A model of *PolynomialTraits_d::ConstructPolynomial*.

PolynomialTraits_d:: Get_coefficient A model of *PolynomialTraits_d::GetCoefficient*.

PolynomialTraits_d:: Get_innermost_coefficient A model of *PolynomialTraits_d::GetInnermostCoefficient*.

PolynomialTraits_d:: Construct_coefficient_const_iterator_range

A model of *PolynomialTraits_d::ConstructCoefficientConstIteratorRange*.

PolynomialTraits_d:: Construct_innermost_coefficient_const_iterator_range

A model of *PolynomialTraits_d::ConstructInnermostCoefficientConstIteratorRange*.

PolynomialTraits_d:: Swap A model of *PolynomialTraits_d::Swap*.

PolynomialTraits_d:: Move A model of *PolynomialTraits_d::Move*.

PolynomialTraits_d:: Degree A model of *PolynomialTraits_d::Degree*.

PolynomialTraits_d:: Total_degree A model of *PolynomialTraits_d::TotalDegree*.

PolynomialTraits_d:: Degree_vector A model of *PolynomialTraits_d::DegreeVector*.

PolynomialTraits_d:: Leading_coefficient A model of *PolynomialTraits_d::LeadingCoefficient*.

PolynomialTraits_d:: Innermost_leading_coefficient

A model of *PolynomialTraits_d::InnermostLeadingCoefficient*.

PolynomialTraits_d:: Canonicalize A model of *PolynomialTraits_d::Canonicalize*.

PolynomialTraits_d:: Differentiate A model of *PolynomialTraits_d::Differentiate*.

PolynomialTraits_d:: Evaluate	A model of *PolynomialTraits_d::Evaluate*.
PolynomialTraits_d:: Evaluate_homogeneous	A model of *PolynomialTraits_d::EvaluateHomogeneous*.
PolynomialTraits_d:: Substitute	A model of *PolynomialTraits_d::Substitute*.
PolynomialTraits_d:: Substitute_homogeneous	A model of *PolynomialTraits_d::SubstituteHomogeneous*.
PolynomialTraits_d:: Is_zero_at	A model of *PolynomialTraits_d::IsZeroAt*.
PolynomialTraits_d:: Is_zero_at_homogeneous	A model of *PolynomialTraits_d::IsZeroAtHomogeneous*.
PolynomialTraits_d:: Sign_at	A model of *PolynomialTraits_d::SignAt*. In case *Innermost_coefficient_type* is not *RealEmbeddable* this is *CGAL::Null_functor*.
PolynomialTraits_d:: Sign_at_homogeneous	A model of *PolynomialTraits_d::SignAtHomogeneous*. In case *Innermost_coefficient_type* is not *RealEmbeddable* this is *CGAL::Null_functor*.
PolynomialTraits_d:: Compare	A model of *PolynomialTraits_d::Compare*. In case *Innermost_coefficient_type* is not *RealEmbeddable* this is *CGAL::Null_functor*.
PolynomialTraits_d:: Univariate_content	In case *PolynomialTraits_d::Coefficient_type* is **not** a model of *UniqueFactorizationDomain*, this is *CGAL::Null_functor*, otherwise this is a model of *PolynomialTraits_d::UnivariateContent*.
PolynomialTraits_d:: Multivariate_content	In case *PolynomialTraits_d::Innermost_coefficient_type* is **not** a model of *UniqueFactorizationDomain*, this is *CGAL::Null_functor*, otherwise this is a model of *PolynomialTraits_d::MultivariateContent*.
PolynomialTraits_d:: Shift	A model of *PolynomialTraits_d::Shift*.
PolynomialTraits_d:: Negate	A model of *PolynomialTraits_d::Negate*.
PolynomialTraits_d:: Invert	A model of *PolynomialTraits_d::Invert*.
PolynomialTraits_d:: Translate	A model of *PolynomialTraits_d::Translate*.
PolynomialTraits_d:: Translate_homogeneous	A model of *PolynomialTraits_d::TranslateHomogeneous*.
PolynomialTraits_d:: Scale	A model of *PolynomialTraits_d::Scale*.
PolynomialTraits_d:: Scale_homogeneous	A model of *PolynomialTraits_d::ScaleHomogeneous*.
PolynomialTraits_d:: Make_square_free	A model of *PolynomialTraits_d::MakeSquareFree*.
PolynomialTraits_d:: Square_free_factorize	In case *PolynomialTraits::Polynomial_d* is not a model of *UniqueFactorizationDomain*, this is of type *CGAL::Null_functor*, otherwise this is a model of *PolynomialTraits_d::SquareFreeFactorize*.

PolynomialTraits_d:: Pseudo_division	A model of *PolynomialTraits_d::PseudoDivision*.
PolynomialTraits_d:: Pseudo_division_remainder	A model of *PolynomialTraits_d::PseudoDivisionRemainder*.
PolynomialTraits_d:: Pseudo_division_quotient	A model of *PolynomialTraits_d::PseudoDivisionQuotient*.
PolynomialTraits_d:: Gcd_up_to_constant_factor	A model of *PolynomialTraits_d::GcdUpToConstantFactor*.
PolynomialTraits_d:: Integral_division_up_to_constant_factor	A model of *PolynomialTraits_d::IntegralDivisionUpToConstantFactor*.
PolynomialTraits_d:: Content_up_to_constant_factor	A model of *PolynomialTraits_d::UnivariateContentUpToConstantFactor*.
PolynomialTraits_d:: Square_free_factorize_up_to_constant_factor	A model of *PolynomialTraits_d::SquareFreeFactorizeUpToConstantFactor*.
PolynomialTraits_d:: Resultant	A model of *PolynomialTraits_d::Resultant*.
PolynomialTraits_d:: Polynomial_subresultants	Either *CGAL::Null_functor* or a model of *PolynomialTraits_d::PolynomialSubresultants*.
PolynomialTraits_d:: Polynomial_subresultants_with_cofactors	Either *CGAL::Null_functor* or a model of *PolynomialTraits_d::PolynomialSubresultants_with_cofactors*.
PolynomialTraits_d:: Principal_subresultants	Either *CGAL::Null_functor* or a model of *PolynomialTraits_d::PrincipalSubresultants*.
PolynomialTraits_d:: Sturm_habicht_sequence	Either *CGAL::Null_functor* or a model of *PolynomialTraits_d::SturmHabichtSequence*.
PolynomialTraits_d:: Sturm_habicht_sequence_with_cofactors	Either *CGAL::Null_functor* or a model of *PolynomialTraits_d::SturmHabichtSequenceWithCofactors*.
PolynomialTraits_d:: Principal_sturm_habicht_sequence	Either *CGAL::Null_functor* or a model of *PolynomialTraits_d::PrincipalSturmHabichtSequence*.

See Also

Polynomial_d . . . . page 261

Has Models

CGAL::Polynomial_traits_d<Polynomial_d> . . . . page 326

PolynomialTraits_d::ConstructPolynomial

Definition

This *AdaptableFunctor* provides several operators to construct objects of type *PolynomialTraits_d::Polynomial_d*.

Refines

AdaptableFunctor
CopyConstructible
DefaultConstructible

Types

typedef PolynomialTraits_d::Polynomial_d

result_type;

Operations

result_type *fo()* Construct the zero polynomial.

result_type *fo(int i)* Construct the constant polynomial equal to *i*.

result_type *fo(PolynomialTraits_d::Innermost_coefficient_type a)*

Construct the constant polynomial equal to *a*.

result_type *fo(PolynomialTraits_d::Coefficient_type a)*

Construct the polynomial equal to *a*.

template < class InputIterator >
result_type *fo(InputIterator begin, InputIterator end)*

Precondition: The value type of *InputIterator* is *PolynomialTraits_d::Coefficient_type*.

The operator constructs the a polynomial from the iterator range, with respect to the outermost variable, x_{d-1}.
The range starts with the coefficient for x_{d-1}^0.
In case the range is empty, the zero polynomial is constructed.

template < class InputIterator >

result_type *fo(InputIterator begin, InputIterator end, bool is_sorted= false)*

Constructs a *Polynomial_d* from a given iterator range of *std::pair< Exponent_vector, PolynomialTraits_d::Innermost_coefficient_type>*.
The optional parameter *is_sorted* indicates whether the given iterator range is already sorted.
Precondition: The value type of *InputIterator* is *std::pair<Exponent_vector, PolynomialTraits_d::Innermost_coefficient_type>*.
Precondition: Each *Exponent_vector* must have size d.
Precondition: All appearing *Exponent_vector*s are different.

See Also

Polynomial_d . . . . page 261
PolynomialTraits_d . . . . page 262

PolynomialTraits_d::GetCoefficient

Concept

Definition

This *AdaptableBinaryFunction* provides access to coefficients of a *PolynomialTraits_d::Polynomial_d*.

Refines

AdaptableBinaryFunction
CopyConstructible
DefaultConstructible

Types

typedef PolynomialTraits_d::Coefficient_type

result_type;

typedef PolynomialTraits_d::Polynomial_d

first_argument_type;

typedef int *second_argument_type;*

Operations

result_type *fo(first_argument_type p, second_argument_type e)*

For given polynomial p this operator returns the coefficient of x_{d-1}^{e}, where x_{d-1} is the outermost variable.

See Also

Polynomial_d . . . . page 261
PolynomialTraits_d . . . . page 262

Concept

PolynomialTraits_d::GetInnermostCoefficient

Definition

For the given *PolynomialTraits_d::Polynomial_d* this *AdaptableBinaryFunction* returns the coefficient of the (multivariate) monomial specified by the given *Exponent_vector*.

Refines

AdaptableBinaryFunction
CopyConstructible
DefaultConstructible

Types

typedef PolynomialTraits_d::Innermost_coefficient_type

result_type;

typedef PolynomialTraits_d::Polynomial_d

first_argument_type;

typedef Exponent_vector

second_argument_type;

Operations

result_type *fo(first_argument_type p, second_argument_type v)*

For given polynomial p this operator returns the innermost coefficient of the monomial corresponding to the given *Exponent_vector* v.

See Also

Polynomial_d . . . . page 261
PolynomialTraits_d . . . . page 262

PolynomialTraits_d::MonomialRepresentation

Concept

Definition

This *Functor* outputs the monomial representation of the given polynomial, that is, it writes all non zero terms of the polynomial as *std::pair<CGAL::Exponent_vector, PolynomialTraits_d::Innermost_coefficient_type>* into the given output iterator.

Refines

Assignable
CopyConstructible
DefaultConstructible

Operations

template<class OutputIterator>
OutputIterator *fo(PolynomialTraits_d::Polynomial_d p, OutputIterator it)*

Writes the monom representation of *p* into the given output iterator *it*.
Precondition: *std::iterator_traits< OutputIterator >::value_type* must be constructible from *std::pair<CGAL::Exponent_vector, PolynomialTraits_d::Innermost_coefficient_type>*.

See Also

Polynomial_d . . . . page 261
PolynomialTraits_d . . . . page 262
PolynomialTraits_d::ConstructPolynomial . . . . page 267

Concept

PolynomialTraits_d::ConstructCoefficientConstIteratorRange

Definition

This *AdaptableUnaryFunction* returns a const iterator range over the coefficients of the given polynomial, with respect to the outermost variable, x_{d-1}. The range starts with the coefficient for x_{d-1}^0.

Refines

AdaptableUnaryFunction
CopyConstructible
DefaultConstructible

Types

typedef std::pair< PolynomialTraits_d::Coefficient_const_iterator, PolynomialTraits_d::Coefficient_const_iterator > *result_type;*

typedef PolynomialTraits_d::Polynomial_d *argument_type;*

Operations

result_type *fo(argument_type p)*

Returns a const iterator range over the coefficients of *p*.

See Also

Polynomial_d . . . . page 261
PolynomialTraits_d . . . . page 262

PolynomialTraits_d::ConstructInnermostCoefficientConstIteratorRange

Concept

Definition

This *AdaptableUnaryFunction* returns a const iterator range over all innermost coefficients of the given polynomial.

Refines

AdaptableUnaryFunction
CopyConstructible
DefaultConstructible

Types

typedef std::pair< PolynomialTraits_d::Innermost_coefficient_const_iterator, PolynomialTraits_d::Innermost_coefficient_const_iterator > *result_type;*

typedef PolynomialTraits_d::Polynomial_d *argument_type;*

Operations

result_type *fo(argument_type p)*

Returns a const iterator range over all innermost coefficients of *p*.

See Also

Polynomial_d . . . page 261
PolynomialTraits_d . . . page 262

Concept

PolynomialTraits_d::Permute

Definition

This *AdaptableFunctor* permutes the variables of the given polynomial with respect to a permutation σ, that is, each monomial $\prod x_i^{e_i}$ will be mapped to the monomial $\prod x_{\sigma(i)}^{e_i}$. The permutation σ is given by the iterator range of length *PolynomialTraits_d::d*, which is supposed to contain the second row of the permutation.

For instance, let p be a polynomial in 4 variables and it is intended to change the order of the variables such that $x_0 \mapsto x_2$, $x_1 \mapsto x_0$, $x_2 \mapsto x_1$ and $x_3 \mapsto x_3$. In this case the iterator range should contain the sequence $[2,0,1,3]$.

Refines

AdaptableFunctor
CopyConstructible
DefaultConstructible

Types

typedef PolynomialTraits_d::Polynomial_d   *result_type;*

Operations

Operations

template<class Input_iterator>
result_type   *fo.operator()(PolynomialTraits_d::Polynomial_d p, Input_iterator begin, Input_iterator end)*

Returns p with interchanged variables as defined by the iterator range.
Precondition: (end-begin == *PolynomialTraits_d::d*)
Precondition: *std::iterator_traits< InputIterator > ::value_type* is convertible to int.
Precondition: The iterator range contains each value in $\{0,\ldots,d-1\}$ exactly once.

See Also

Polynomial_d . . . . page 261
PolynomialTraits_d . . . . page 262

PolynomialTraits_d::Swap

Concept

Definition

This *AdaptableFunctor* swaps two variables of a multivariate polynomial.

Refines

AdaptableFunctor
CopyConstructible
DefaultConstructible

Types

typedef PolynomialTraits_d::Polynomial_d	*result_type;*

Operations

result_type	*fo(PolynomialTraits_d::Polynomial_d p, int i, int j)*

Returns polynomial p with interchanged variables x_i,x_j.
Precondition: $0 \leq i < d$
Precondition: $0 \leq j < d$

See Also

Polynomial_d . . . . page 261
PolynomialTraits_d . . . . page 262

PolynomialTraits_d::Move

Definition

This *AdaptableFunctor* moves a variable at position i to a new position j. The relative order of the other variables is preserved, that is, the variables between x_i and x_j (including x_j) are moved by one position while x_i is moved to the former position of x_j.

This function may be used to make a certain variable the outer most variable.

Refines

AdaptableFunctor
CopyConstructible
DefaultConstructible

Types

typedef PolynomialTraits_d::Polynomial_d *result_type;*

Operations

result_type *fo(PolynomialTraits_d::Polynomial_d, int i, int j)*

This function moves the variable at position i to its new position j and returns the new polynomial. The relative order of the other variables is preserved.
Precondition: $0 \leq i < d$
Precondition: $0 \leq j < d$

See Also

Polynomial_d . . . . page 261
PolynomialTraits_d . . . . page 262

PolynomialTraits_d::Degree

Concept

Definition

This *AdaptableUnaryFunction* computes the degree of a *PolynomialTraits_d::Polynomial_d* with respect to a certain variable.

The degree of p with respect to a certain variable x_i, is the highest power e of x_i such that the coefficient of x_i^e in p is not zero.
For instance the degree of $p = x_0^2 x_1^3 + x_1^4$ with respect to x_1 is 4.

The degree of the zero polynomial is set to 0. From the mathematical point of view this should be $-infinity$, but this would imply an inconvenient return type.

Refines

AdaptableUnaryFunction
CopyConstructible
DefaultConstructible

Types

typedef int	*result_type;*
typedef PolynomialTraits_d::Polynomial_d	*argument_type;*

Operations

result_type *fo(argument_type p)*

Computes the degree of p with respect to the outermost variable x_{d-1}.

result_type *fo(argument_type p, int i)*

Computes the degree of p with respect to variable x_i.
Precondition: $0 \leq i < d$

See Also

Polynomial_d . . . page 261
PolynomialTraits_d . . . page 262
PolynomialTraits_d::TotalDegree . . . page 278
PolynomialTraits_d::DegreeVector . . . page 279

Concept

PolynomialTraits_d::TotalDegree

Definition

This *AdaptableUnaryFunction* computes the total degree of a *PolynomialTraits_d::Polynomial_d*.

Given a (multivariate) monomial the sum of all appearing exponents is the total degree of this monomial. The total degree of a polynomial p is the maximum of the total degrees of all appearing (multivariate) monomials in p.
For instance the total degree of $p = x_0^2 x_1^3 + x_1^4$ is 5.

The total degree of the zero polynomial is set to 0. From the mathematical point of view this should be $-inf$, but this would imply an inconvenient return type.

Refines

AdaptableUnaryFunction
CopyConstructible
DefaultConstructible

Types

typedef int	*result_type;*
typedef PolynomialTraits_d::Polynomial_d	*argument_type;*

Operations

result_type *fo(argument_type p)*

Computes the total degree of p.

See Also

Polynomial_d . . . . . . page 261
PolynomialTraits_d . . . . . . page 262
PolynomialTraits_d::Degree . . . . . . page 277
PolynomialTraits_d::DegreeVector . . . . . . page 279

PolynomialTraits_d::DegreeVector

Concept

Definition

For a given *PolynomialTraits_d::Polynomial_d* p this *AdaptableUnaryFunction* returns the degree vector, that is, it returns the exponent vector of the monomial of highest order in p, where the monomial order is the lexicographic order giving outer variables a higher priority. In particular, this is the monomial that belongs to the innermost leading coefficient of p.

Refines

AdaptableUnaryFunction
CopyConstructible
DefaultConstructible

Types

typedef Exponent_vector	*result_type;*
typedef PolynomialTraits_d::Polynomial_d	*argument_type;*

Operations

result_type *fo(argument_type p)*

Returns the degree vector.

See Also

Polynomial_d . . . . page 261
PolynomialTraits_d . . . . page 262
PolynomialTraits_d::Degree . . . . page 277
PolynomialTraits_d::TotalDegree . . . . page 278
PolynomialTraits_d::InnermostLeadingCoefficient . . . . page 281

Concept

PolynomialTraits_d::LeadingCoefficient

Definition

This *AdaptableUnaryFunction* computes the leading coefficient of a *PolynomialTraits_d::Polynomial_d*.

Refines

AdaptableUnaryFunction
CopyConstructible
DefaultConstructible

Types

typedef PolynomialTraits_d::Coefficient_type	*result_type;*
typedef PolynomialTraits_d::Polynomial_d	*argument_type;*

Operations

result_type *fo(argument_type p)*

Computes the leading coefficient of p with respect to the outermost variable x_{d-1}.

See Also

Polynomial_d . . . . page 261
PolynomialTraits_d . . . . page 262

PolynomialTraits_d::InnermostLeadingCoefficient

Definition

This *AdaptableUnaryFunction* computes the innermost leading coefficient of a *PolynomialTraits_d::Polynomial_d* p. The innermost leading coefficient is recursively defined as the innermost leading coefficient of the leading coefficient of p. In case p is univariate it coincides with the leading coefficient.

Refines

AdaptableUnaryFunction
CopyConstructible
DefaultConstructible

Types

typedef PolynomialTraits_d::Innermost_coefficient_type
result_type;

typedef PolynomialTraits_d::Polynomial_d *argument_type;*

Operations

result_type *fo(argument_type p)*

Computes the innermost leading coefficient of p.

See Also

Polynomial_d . . . . page 261
PolynomialTraits_d . . . . page 262

PolynomialTraits_d::Canonicalize

Definition

For a given polynomial p this *AdaptableUnaryFunction* computes the unique representative of the set

$$\mathcal{P} := \{q \mid \lambda * q = p \ for \ some \ \lambda \in R\},$$

where R is the base of the polynomial ring.

In case *PolynomialTraits::Innermost_coefficient_type* is a model of *Field*, the computed polynomial is the *monic* polynomial in $\mathcal{P}$, that is, the innermost leading coefficient equals one.
In case *PolynomialTraits::Innermost_coefficient_type* is a model of *UniqueFactorizationDomain*, the computed polynomial is the one with a multivariate content of one.
For all other cases the notion of uniqueness is up to the concrete model.

Note that the computed polynomial has the same zero set as the given one.

Refines

AdaptableUnaryFunction
CopyConstructible
DefaultConstructible

Types

typedef PolynomialTraits_d::Polynomial_d	*result_type;*
typedef PolynomialTraits_d::Polynomial_d	*argument_type;*

Operations

result_type *fo(first_argument_type p)*

Returns the canonical representative of p.

See Also

Polynomial_d .... page 261
PolynomialTraits_d .... page 262

PolynomialTraits_d::Differentiate

Concept

Definition

This *AdaptableUnaryFunction* computes the derivative of a *PolynomialTraits_d::Polynomial_d* with respect to one variable.

Refines

AdaptableUnaryFunction
CopyConstructible
DefaultConstructible

Types

typedef PolynomialTraits_d::Polynomial_d	*result_type;*
typedef PolynomialTraits_d::Polynomial_d	*argument_type;*

Operations

result_type *fo(argument_type p)*

Returns p', with respect to the outermost variable.

result_type *fo(argument_type p, int i)*

Returns p', with respect to variable x_i.
Precondition: $0 \leq i < d$

See Also

Polynomial_d . . . . page 261
PolynomialTraits_d . . . . page 262

Concept

PolynomialTraits_d::Evaluate

Definition

This *AdaptableBinaryFunction* evaluates *PolynomialTraits_d::Polynomial_d* with respect to one variable.

Refines

AdaptableBinaryFunction
CopyConstructible
DefaultConstructible

Types

typedef PolynomialTraits_d::Coefficient_type *result_type;*
typedef PolynomialTraits_d::Polynomial_d *first_argument_type;*
typedef PolynomialTraits_d::Coefficient_type *second_argument_type;*

Operations

result_type *fo(first_argument_type p, second_argument_type x)*

Returns $p(x)$, with respect to the outermost variable.

See Also

Polynomial_d . . . . page 261
PolynomialTraits_d . . . . page 262

PolynomialTraits_d::EvaluateHomogeneous

Concept

Definition

This *AdaptableFunctor* provides evaluation of a *PolynomialTraits_d::Polynomial_d* interpreted as a homogeneous polynomial **in one variable**.
For instance the polynomial $p = 5x^2y^3 + y$ is interpreted as the homogeneous polynomial $p[x](u,v) = 5x^2u^3 + uv^2$ and evaluated as such.

Refines

AdaptableFunctor
CopyConstructible
DefaultConstructible

Types

typedef PolynomialTraits_d::Coefficient_type

result_type;

Operations

result_type *fo.operator()(PolynomialTraits_d::Polynomial_d p, PolynomialTraits_d::Coefficient_type u, PolynomialTraits_d::Coefficient_type v)*

Returns $p(u,v)$, with respect to the outermost variable.

See Also

Polynomial_d . . . . page 261
PolynomialTraits_d . . . . page 262

Concept

PolynomialTraits_d::Substitute

Definition

This *Functor* substitutes all variables of a given multivariate *PolynomialTraits_d::Polynomial_d* by the values given in the iterator range, where begin refers the value for the innermost variable.

Refines

Assignable
CopyConstructible
DefaultConstructible

Types

Note that the *result_type* is the coercion type of the value type of the given iterator range and *PolynomialTraits_d::Innermost_coefficient_type*. In particular *std::iterator_traits<Input_iterator>::value_type* must be *ExplicitInteroperable* with *PolynomialTraits_d::Innermost_coefficient_type*. Hence, it can not be provided as a public type in advance.

Operations

template<class Input_iterator>
result_type *fo.operator()(PolynomialTraits_d::Polynomial_d p, Input_iterator begin, Input_iterator end)*

Substitutes each variable of p by the values given in the iterator range, where begin refers to the innermost variable x_0.
Precondition: (end-begin == *PolynomialTraits_d::d*)

See Also

Polynomial_d .... page 261
PolynomialTraits_d .... page 262
PolynomialTraits_d::SubstituteHomogeneous .... page 287
CoercionTraits .... page **??**

PolynomialTraits_d::SubstituteHomogeneous

Concept

Definition

This *Functor* substitutes all variables of a given multivariate *PolynomialTraits_d::Polynomial_d p* by the values given in the iterator range, where begin refers the value for the innermost variable. In contrast to *PolynomialTraits_d::Substitute* the given polynomial p is interpreted as a homogeneous polynomial. Hence the iterator range is required to be of length *PolynomialTraits_d::d+1*.
For instance the polynomial $p(x_0,x_1) = x_0^2x_1^3 + x_1^4$ is interpreted as the homogeneous polynomial $p(x_0,x_1,w) = x_0^2x_1^3 + x_1^4w^1$.

Refines

Assignable
CopyConstructible
DefaultConstructible

Types

Note that the *result_type* is the coercion type of the value type of the given iterator range and *PolynomialTraits_d::Innermost_coefficient_type*. In particular *std::iterator_traits<Input_iterator>::value_type* must be *ExplicitInteroperable* with *PolynomialTraits_d::Innermost_coefficient_type*. Hence, it can not be provided as a public type in advance.

Operations

template<class Input_iterator>
result_type *fo.operator()(PolynomialTraits_d::Polynomial_d p, Input_iterator begin, Input_iterator end)*

Substitutes each variable of p by the values given in the iterator range, where p is interpreted as a homogeneous polynomial in all variables. The begin iterator refers to the innermost variable x_0.
Precondition: (end-begin == *PolynomialTraits_d::d*)+1

See Also

Polynomial_d . . . . page 261
PolynomialTraits_d . . . . page 262
PolynomialTraits_d::Substitute . . . . page 286
CoercionTraits . . . . page **??**

Concept

PolynomialTraits_d::IsZeroAt

Definition

This *AdaptableFunctor* returns whether a *PolynomialTraits_d::Polynomial_d p* is zero at a given Cartesian point, which is represented as an iterator range.

Refines

AdaptableFunctor
CopyConstructible
DefaultConstructible

Types

typedef bool *result_type;*

Operations

template <class InputIterator>
result_type *fo.operator()(PolynomialTraits_d::Polynomial_d p, InputIterator begin, InputIterator end)*

Computes whether *p* is zero at the Cartesian point given by the iterator range, where *begin* is referring to the innermost variable.
Precondition: (end-begin == *PolynomialTraits_d::d*)
Precondition: *std::iterator_traits< InputIterator >::value_type* is *ExplicitInteroperable* with *PolynomialTraits_d::Innermost_coefficient_type*.

See Also

Polynomial_d . . . . page 261
PolynomialTraits_d . . . . page 262

PolynomialTraits_d::IsZeroAtHomogeneous

Concept

Definition

This *AdaptableFunctor* returns whether a *PolynomialTraits_d::Polynomial_d p* is zero at a given homogeneous point, which is given by an iterator range.

The polynomial is interpreted as a homogeneous polynomial in all variables.
For instance the polynomial $p(x_0,x_1) = x_0^2 x_1^3 + x_1^4$ is interpreted as the homogeneous polynomial $p(x_0,x_1,w) = x_0^2 x_1^3 + x_1^4 w^1$.

Refines

AdaptableFunctor
CopyConstructible
DefaultConstructible

Types

typedef bool	*result_type;*

Operations

template <class InputIterator>
result_type *fo.operator()(PolynomialTraits_d::Polynomial_d p, InputIterator begin, InputIterator end)*

Computes whether *p* is zero at the homogeneous point given by the iterator range, where *begin* is referring to the innermost variable.
Precondition: (end-begin==*PolynomialTraits_d::d*+1)
Precondition: *std::iterator_traits< InputIterator >::value_type* is *ExplicitInteroperable* with *PolynomialTraits_d::Innermost_coefficient_type*.

See Also

Polynomial_d . . . . page 261
PolynomialTraits_d . . . . page 262

Concept

PolynomialTraits_d::SignAt

Definition

This *AdaptableFunctor* returns the sign of a *PolynomialTraits_d::Polynomial_d p* at given Cartesian point represented as an iterator range.

This functor is well defined if *PolynomialTraits_d::Innermost_coefficient_type* is *RealEmbeddable*.

Refines

AdaptableFunctor
CopyConstructible
DefaultConstructible

Types

typedef CGAL::Sign *result_type;*

Operations

template <class InputIterator>
result_type *fo.operator()(PolynomialTraits_d::Polynomial_d p, InputIterator begin, InputIterator end)*

Returns the sign of *p* at the given Cartesian point, where *begin* is referring to the innermost variable.
Precondition: (end-begin == *PolynomialTraits_d::d*)
Precondition: *std::iterator_traits< InputIterator >::value_type* is *ExplicitInteroperable* with *PolynomialTraits_d::Innermost_coefficient_type*.

See Also

Polynomial_d . . . . page 261
PolynomialTraits_d . . . . page 262

PolynomialTraits_d::SignAtHomogeneous

Concept

Definition

This *AdaptableFunctor* returns the sign of a *PolynomialTraits_d::Polynomial_d* p at a given homogeneous point, which is given by an iterator range.

The polynomial is interpreted as a homogeneous polynomial in all variables.
For instance the polynomial $p(x_0,x_1) = x_0^2x_1^3 + x_1^4$ is interpreted as the homogeneous polynomial $p(x_0,x_1,w) = x_0^2x_1^3 + x_1^4w^1$.

This functor is well defined if *PolynomialTraits_d::Innermost_coefficient_type* is *RealEmbeddable*.

Refines

AdaptableFunctor
CopyConstructible
DefaultConstructible

Types

typedef CGAL::Sign	*result_type;*

Operations

template <class InputIterator>
result_type *fo.operator()(PolynomialTraits_d::Polynomial_d p, InputIterator begin, InputIterator end)*

Returns the sign of p at the given homogeneous point, where *begin* is referring to the innermost variable.
Precondition: (end-begin==*PolynomialTraits_d::d*+1)
Precondition: *std::iterator_traits< InputIterator >::value_type* is *ExplicitInteroperable* with *PolynomialTraits_d::Innermost_coefficient_type*.

See Also

Polynomial_d . . . . page 261
PolynomialTraits_d . . . . page 262

Concept

PolynomialTraits_d::Compare

Definition

This *AdaptableBinaryFunction* compares two polynomials, with respect to the lexicographic order with preference to the outermost variable.

This functor is well defined if *PolynomialTraits_d::Innermost_coefficient_type* is *RealEmbeddable*.

Refines

AdaptableBinaryFunction
CopyConstructible
DefaultConstructible

Types

typedef CGAL::Comparison_result	*result_type;*
typedef PolynomialTraits_d::Polynomial_d	*first_argument_type;*
typedef PolynomialTraits_d::Polynomial_d	*second_argument_type;*

Operations

result_type *fo(first_argument_type f, second_argument_type g)*

Compares two polynomials.

See Also

Polynomial_d . . . . page 261
PolynomialTraits_d . . . . page 262

PolynomialTraits_d::UnivariateContent

Concept

Definition

This *AdaptableUnaryFunction* computes the content of a *PolynomialTraits_d::Polynomial_d* with respect to the univariate (recursive) view on the polynomial, that is, it computes the gcd of all coefficients with respect to one variable.

This functor is well defined if *PolynomialTraits_d::Coefficient_type* is a *Field* or a *UniqueFactorizationDomain*.

Refines

AdaptableUnaryFunction
CopyConstructible
DefaultConstructible

Types

typedef PolynomialTraits_d::Coefficient_type *result_type;*

typedef PolynomialTraits_d::Polynomial_d *argument_type;*

Operations

result_type *fo(argument_type p)*

Computes the content of p with respect to the outermost variable x_{d-1}.

See Also

Polynomial_d . . . . page 261
PolynomialTraits_d . . . . page 262

Concept

PolynomialTraits_d::MultivariateContent

Definition

This *AdaptableUnaryFunction* computes the content of a *PolynomialTraits_d::Polynomial_d* with respect to the symmetric view on the polynomial, that is, it computes the gcd of all innermost coefficients.

This functor is well defined if *PolynomialTraits_d::Innermost_coefficient_type* is a *Field* or a *UniqueFactorizationDomain*.

Refines

AdaptableUnaryFunction
CopyConstructible
DefaultConstructible

Types

typedef PolynomialTraits_d::Innermost_coefficient_type
result_type;

typedef PolynomialTraits_d::Polynomial_d *argument_type;*

Operations

result_type *fo(argument_type p)*

Computes the *gcd* of all innermost coefficients of *p*.

See Also

Polynomial_d .... page 261
PolynomialTraits_d .... page 262

PolynomialTraits_d::SquareFreeFactorize

Definition

This *Functor* computes a square-free factorization of a *PolynomialTraits_d::Polynomial_d*.

A polynomial p is factored into square-free and pairwise coprime non-constant factors g_i with multiplicities m_i and a constant factor a, such that $p = a \cdot g_1^{m_1} \cdot \ldots \cdot g_n^{m_n}$.

The pairs (g_i, m_i) are written into the given output iterator.

This functor is well defined if *PolynomialTraits_d::Polynomial_d* is a *UniqueFactorizationDomain*.

Refines

Assignable
CopyConstructible
DefaultConstructible

Operations

template<class OutputIterator>
OutputIterator *fo.operator()(PolynomialTraits_d::Polynomial_d p, OutputIterator it, PolynomialTraits_d::Innermost_coefficient_type& a)*

Computes the square-free factorization of p and returns the past-the-end iterator of the written range.
Precondition: *std::iterator_traits< OutputIterator > ::value_type* must be constructible from *std::pair< PolynomialTraits_d::Polynomial_d,int>*.

template<class OutputIterator>
OutputIterator *fo(PolynomialTraits_d::Polynomial_d p, OutputIterator it)*

As the first operator, just not computing the factor a.

See Also

Polynomial_d . . . . page 261
PolynomialTraits_d . . . . page 262
PolynomialTraits_d::SquareFreeFactorizeUpToConstantFactor . . . . page 304
PolynomialTraits_d::MakeSquareFree . . . . page 296
PolynomialTraits_d::IsSquareFree . . . . page 297

Concept

PolynomialTraits_d::MakeSquareFree

Definition

This *AdaptableBinaryFunction* computes the square-free part of a polynomial of type *PolynomialTraits_d::Polynomial_d up to a constant factor*.

A polynomial p can be factored into square-free and pairwise coprime non-constant factors g_i with multiplicities m_i and a constant factor a, such that $p = a \cdot g_1^{m_1} \cdot \ldots \cdot g_n^{m_n}$, where all g_i are canonicalized.

Given this decomposition, the square free part is defined as the product $g_1 \cdot \ldots \cdot g_n$, which is computed by this functor.

Refines

AdaptableUnaryFunction
CopyConstructible
DefaultConstructible

Types

typedef PolynomialTraits_d::Polynomial_d	*result_type;*
typedef PolynomialTraits_d::Polynomial_d	*argument_type;*

Operations

result_type *fo(argument_type p)*

Returns the square-free part of p.

See Also

Polynomial_d . . . . page 261
PolynomialTraits_d . . . . page 262
PolynomialTraits_d::Canonicalize . . . . page 282
PolynomialTraits_d::SquareFreeFactorize . . . . page 295
PolynomialTraits_d::IsSquareFree . . . . page 297

PolynomialTraits_d::IsSquareFree

Definition

This *AdaptableBinaryFunction* computes whether the given a polynomial of type *PolynomialTraits_d::Polynomial_d* is square free.

Note that this statement does cover constant factors, i.e., whether the multivariate content contains a square.

Refines

AdaptableUnaryFunction
CopyConstructible
DefaultConstructible

Types

typedef bool	*result_type;*
typedef PolynomialTraits_d::Polynomial_d	*argument_type;*

Operations

result_type	*fo(argument_type p)*

Returns whether the p is square free.

See Also

Concept

PolynomialTraits_d::PseudoDivision

Definition

This *AdaptableFunctor* computes the *pseudo division* of two polynomials f and g.

Given f and $g \neq 0$ this functor computes quotient q and remainder r such that $D \cdot f = g \cdot q + r$ and $degree(r) < degree(g)$, where $D = leading\_coefficient(g)^{max(0,degree(f)-degree(g)+1)}$

This functor is useful if the regular division is not available, which is the case if *PolynomialTraits_d::Coefficient_type* is not a *Field*. Hence in general it is not possible to invert the leading coefficient of g. Instead f is extended by D allowing integral divisions in the internal computation.

Refines

AdaptableFunctor
CopyConstructible
DefaultConstructible

Types

typedef void	*result_type;*

Operations

result_type   *fo.operator()(PolynomialTraits_d::Polynomial_d f, PolynomialTraits_d::Polynomial_d g, PolynomialTraits_d::Polynomial_d & q, PolynomialTraits_d::Polynomial_d & r, PolynomialTraits_d::Coefficient_type & D)*

Computes the pseudo division with respect to the outermost variable x_{d-1}.

See Also

PolynomialTraits_d::PseudoDivisionQuotient

Concept

Definition

This *AdaptableBinaryFunction* computes the quotient of the *pseudo division* of two polynomials f and g.

Given f and $g \neq 0$ one can compute quotient q and remainder r such that $D \cdot f = g \cdot q + r$ and $degree(r) < degree(g)$, where $D = leading\_coefficient(g)^{max(0, degree(f) - degree(g) + 1)}$

This functor computes q.

Refines

AdaptableBinaryFunction
CopyConstructible
DefaultConstructible

Types

typedef PolynomialTraits_d::Polynomial_d	*result_type;*
typedef PolynomialTraits_d::Polynomial_d	*first_argument_type;*
typedef PolynomialTraits_d::Polynomial_d	*second_argument_type;*

Operations

result_type *fo(first_argument_type f, second_argument_type g)*

Returns the quotient q of the pseudo division of f and g with respect to the outermost variable x_{d-1}.

See Also

Polynomial_d . . . . page 261
PolynomialTraits_d . . . . page 262
PolynomialTraits_d::PseudoDivision . . . . page 298
PolynomialTraits_d::PseudoDivisionRemainder . . . . page 300
PolynomialTraits_d::PseudoDivisionQuotient . . . . page 299

PolynomialTraits_d::PseudoDivisionRemainder

Definition

This *AdaptableBinaryFunction* computes the remainder of the *pseudo division* of two polynomials f and g.

Given f and $g \neq 0$ one can compute quotient q and remainder r such that $D \cdot f = g \cdot q + r$ and $degree(r) < degree(g)$, where $D = leading\_coefficient(g)^{max(0, degree(f) - degree(g) + 1)}$

This functor computes r.

Refines

AdaptableBinaryFunction
CopyConstructible
DefaultConstructible

Types

typedef PolynomialTraits_d::Polynomial_d	*result_type;*
typedef PolynomialTraits_d::Polynomial_d	*first_argument_type;*
typedef PolynomialTraits_d::Polynomial_d	*second_argument_type;*

Operations

result_type *fo(first_argument_type f, second_argument_type g)*

Returns the remainder r of the pseudo division of f and g with respect to the outermost variable x_{d-1}.

See Also

Polynomial_d . . . . page 261
PolynomialTraits_d . . . . page 262
PolynomialTraits_d::PseudoDivision . . . . page 298
PolynomialTraits_d::PseudoDivisionRemainder . . . . page 300
PolynomialTraits_d::PseudoDivisionQuotient . . . . page 299

PolynomialTraits_d::GcdUpToConstantFactor

Concept

Definition

This *AdaptableBinaryFunction* computes the *gcd up to a constant factor (utcf)* of two polynomials of type *PolynomialTraits_d::Polynomial_d*.

In case the base ring R (*PolynomialTraits_d::Innermost_coefficient_type*) is not a *UniqueFactorizationDomain* or not a *Field* the polynomial ring $R[x_0,\ldots,x_{d-1}]$ (*PolynomialTraits_d::Polynomial_d*) may not possesses greatest common divisors. However, since R is an integral domain one can consider its quotient field $Q(R)$ for which *gcd*s of polynomials exist.

This functor computes $gcd\_utcf(f,g) = D * gcd(f,g)$, for some $D \in R$ such that $gcd\_utcf(f,g) \in R[x_0,\ldots,x_{d-1}]$. Hence, $gcd\_utcf(f,g)$ may not be a divisor of f and g in $R[x_0,\ldots,x_{d-1}]$.

Refines

AdaptableBinaryFunction
CopyConstructible
DefaultConstructible

Types

typedef PolynomialTraits_d::Polynomial_d	*result_type;*
typedef PolynomialTraits_d::Polynomial_d	*first_argument_type;*
typedef PolynomialTraits_d::Polynomial_d	*second_argument_type;*

Operations

result_type	*fo(first_argument_type f, second_argument_type g)*

Computes $gcd(f,g)$ up to a constant factor.

See Also

Polynomial_d . . . . . page 261
PolynomialTraits_d . . . . . page 262
PolynomialTraits_d::IntegralDivisionUpToConstantFactor . . . . . page 302
PolynomialTraits_d::UnivariateContentUpToConstantFactor . . . . . page 303
PolynomialTraits_d::SquareFreeFactorizeUpToConstantFactor . . . . . page 304

PolynomialTraits_d::IntegralDivisionUpToConstantFactor

Definition

This *AdaptableBinaryFunction* computes the integral division of two polynomials of type *PolynomialTraits_d::Polynomial_d up to a constant factor (utcf)* .

Precondition: g divides f in $Q(R)[x_0, \ldots, x_{d-1}]$, where $Q(R)$ is the quotient field of the base ring R, *PolynomialTraits_d::Innermost_coefficient_type*.

Refines

AdaptableBinaryFunction
CopyConstructible
DefaultConstructible

Types

typedef PolynomialTraits_d::Polynomial_d	*result_type;*
typedef PolynomialTraits_d::Polynomial_d	*first_argument_type;*
typedef PolynomialTraits_d::Polynomial_d	*second_argument_type;*

Operations

result_type *fo(first_argument_type f, second_argument_type g)*

Computes f/g up to a constant factor.

See Also

Polynomial_d . . . page 261
PolynomialTraits_d . . . page 262
PolynomialTraits_d::GcdUpToConstantFactor . . . page 301

PolynomialTraits_d::UnivariateContentUpToConstantFactor

Concept

Definition

This *AdaptableUnaryFunction* computes the content of a *PolynomialTraits_d::Polynomial_d* with respect to the univariate (recursive) view on the polynomial *up to a constant factor (utcf)*, that is, it computes the *gcd_utcf* of all coefficients with respect to one variable.

Remark: This is called *UnivariateContentUpToConstantFactor* for symmetric reasons with respect to *PolynomialTraits_d::UnivariateContent* and *PolynomialTraits_d::MultivariateContent*. However, a concept *PolynomialTraits_d::MultivariateContentUpToConstantFactor* does not exist since the result is trivial.

Refines

AdaptableUnaryFunction
CopyConstructible
DefaultConstructible

Types

typedef PolynomialTraits_d::Coefficient_type	*result_type;*
typedef PolynomialTraits_d::Polynomial_d	*argument_type;*

Operations

result_type *fo(first_argument_type p)*

Computes the content *up to a constant factor* of p with respect to the outermost variable x_{d-1}.

See Also

Polynomial_d . . . . . page 261
PolynomialTraits_d . . . . . page 262
PolynomialTraits_d::GcdUpToConstantFactor . . . . . page 301

PolynomialTraits_d::SquareFreeFactorizeUpToConstantFactor

Definition

This *AdaptableFunctor* computes a square-free factorization *up to a constant factor (utcf)* of a *PolynomialTraits_d::Polynomial_d*.

A polynomial p is factored into square-free and pairwise coprime non-constant factors g_i with multiplicities m_i, such that $a \cdot p = g_1^{m_1} \cdot \ldots \cdot g_n^{m_n}$, where a is some constant factor.

The pairs (g_i, m_i) are written into the given output iterator.
The constant factor a is not computed.

This functor is well defined even though *PolynomialTraits_d::Innermost_coefficient_type* may not be a *UniqueFactorizationDomain*.

Refines

Assignable
CopyConstructible
DefaultConstructible

Operations

template<class OutputIterator>
OutputIterator *fo(PolynomialTraits_d::Polynomial_d p, OutputIterator it)*

Computes the square-free factorization of p and returns the past-the-end iterator of the written range.
Precondition: *std::iterator_traits< OutputIterator > ::value_type* must be constructible from *std::pair< PolynomialTraits_d::Polynomial_d,int>*.

See Also

Polynomial_d . . . . page 261
PolynomialTraits_d . . . . page 262
PolynomialTraits_d::SquareFreeFactorize . . . . page 295

PolynomialTraits_d::Shift

Definition

This *AdaptableBinaryFunction* multiplies a *PolynomialTraits_d::Polynomial_d* by the given power of the specified variable.

This functor is provided for efficiency reasons, since multiplication by some variable will in general correspond to a shift of coefficients in the internal representation.

Refines

AdaptableBinaryFunction
CopyConstructible
DefaultConstructible

Types

typedef PolynomialTraits_d::Polynomial_d	*result_type;*
typedef PolynomialTraits_d::Polynomial_d	*first_argument_type;*
typedef int	*second_argument_type;*

Operations

result_type *fo(first_argument_type p, second_argument_type e)*

Returns $p * x_{d-1}^{e}$.
Precondition: $0 \leq e$

result_type *fo(first_argument_type p, second_argument_type e, int i)*

Returns $p * x_{i}^{e}$.
Precondition: $0 \leq e$
Precondition: $0 \leq i < d$

See Also

Polynomial_d . . . . page 261
PolynomialTraits_d . . . . page 262

Concept

PolynomialTraits_d::Negate

Definition

This *AdaptableUnaryFunction* computes $p(-x)$ for a given polynomial p.

Note that this functor operates on the polynomial in the univariate view, that is, the polynomial is considered as a univariate polynomial in one specific variable.

This functor is provided for efficiency reasons, since this operation just flips the sign of all odd coefficients with respect to the specified variable.

Refines

AdaptableUnaryFunction
CopyConstructible
DefaultConstructible

Types

typedef PolynomialTraits_d::Polynomial_d	*result_type;*
typedef PolynomialTraits_d::Polynomial_d	*argument_type;*

Operations

result_type *fo(argument_type p)*

Returns $p(-x)$, with respect to the outermost variable.

result_type *fo(argument_type p, int i)*

Returns $p(-x)$, with respect to variable x_i.
Precondition: $0 \leq i < d$

See Also

Polynomial_d . . . . page 261
PolynomialTraits_d . . . . page 262

PolynomialTraits_d::Invert

Concept

Definition

This *AdaptableUnaryFunction* inverts one variable in a given *PolynomialTraits_d::Polynomial_d*, that is, for a given polynomial p it computes $x^{degree(p)}p(1/x)$.

Note that this functor operates on the polynomial in the univariate view, that is, the polynomial is considered as a univariate polynomial in one specific variable.

This functor is provided for efficiency reasons, since this operation just inverts the order of the coefficients with respect to the specified variable.

Refines

AdaptableUnaryFunction
CopyConstructible
DefaultConstructible

Types

typedef PolynomialTraits_d::Polynomial_d	*result_type;*
typedef PolynomialTraits_d::Polynomial_d	*argument_type;*

Operations

result_type *fo(argument_type p)*

Returns $x^{degree(p)}p(1/x)$, where x refers to the outermost variable x_{d-1}.

result_type *fo(argument_type p, int i)*

Return $x^{degree(p,i)}p(1/x)$, where x refers to the variable x_i.
Precondition: $0 \leq i < d$

See Also

Polynomial_d . . . . page 261
PolynomialTraits_d . . . . page 262

Concept

PolynomialTraits_d::Translate

Definition

This *AdaptableBinaryFunction* translates a *PolynomialTraits_d::Polynomial_d* with respect to one variable, that is, for a given polynomial p and constant c it computes $p(x+c)$.

Note that this functor operates on the polynomial in the univariate view, that is, the polynomial is considered as a univariate polynomial in one specific variable.

Refines

AdaptableBinaryFunction
CopyConstructible
DefaultConstructible

Types

typedef PolynomialTraits_d::Polynomial_d *result_type;*
typedef PolynomialTraits_d::Polynomial_d *first_argument_type;*
typedef PolynomialTraits_d::Innermost_coefficient_type *second_argument_type;*

Operations

result_type *fo(first_argument_type p, second_argument_type c)*

Returns $p(x+c)$, with respect to the outermost variable.

result_type *fo(first_argument_type p, second_argument_type c, int i)*

Same as first operator but for variable x_i.
Precondition: $0 \leq i < d$

See Also

Polynomial_d . . . . page 261
PolynomialTraits_d . . . . page 262

Concept

PolynomialTraits_d::TranslateHomogeneous

Definition

Given numerator a and denominator b this *AdaptableFunctor* translates a *PolynomialTraits_d::Polynomial_d* p with respect to one variable by a/b, that is, it computes $b^{degree(p)} \cdot p(x+a/b)$.

Note that this functor operates on the polynomial in the univariate view, that is, the polynomial is considered as a univariate homogeneous polynomial in one specific variable.

Refines

AdaptableFunctor
CopyConstructible
DefaultConstructible

Types

typedef PolynomialTraits_d::Polynomial_d *result_type;*

Operations

result_type *fo.operator()(PolynomialTraits_d::Polynomial_d p, PolynomialTraits_d::Innermost_coefficient_type a, PolynomialTraits_d::Innermost_coefficient_type b)*

Returns $b^{degree(p)} \cdot p(x+a/b)$, with respect to the outermost variable.

result_type *fo.operator()(PolynomialTraits_d::Polynomial_d p, PolynomialTraits_d::Innermost_coefficient_type a, PolynomialTraits_d::Innermost_coefficient_type b, int i)*

Same as first operator but for variable x_i.
Precondition: $0 \leq i < d$

See Also

Polynomial_d . . . . . . page 261
PolynomialTraits_d . . . . . . page 262

Concept

PolynomialTraits_d::Scale

Definition

Given a constant c this *AdaptableBinaryFunction* scales a *PolynomialTraits_d::Polynomial_d* p with respect to one variable, that is, it computes $p(c \cdot x)$.

Note that this functor operates on the polynomial in the univariate view, that is, the polynomial is considered as a univariate polynomial in one specific variable.

Refines

AdaptableBinaryFunction
CopyConstructible
DefaultConstructible

Types

typedef PolynomialTraits_d::Polynomial_d *result_type;*
typedef PolynomialTraits_d::Polynomial_d *first_argument_type;*
typedef PolynomialTraits_d::Innermost_coefficient_type
second_argument_type;

Operations

result_type *fo(first_argument_type p, second_argument_type c)*

Returns $p(c \cdot x)$, with respect to the outermost variable.

result_type *fo(first_argument_type p, second_argument_type c, int i)*

Same as first operator but for variable x_i.
Precondition: $0 \leq i < d$

See Also

Polynomial_d . . . . page 261
PolynomialTraits_d . . . . page 262

Concept

PolynomialTraits_d::ScaleHomogeneous

Definition

Given a numerator a and a denominator b this *AdaptableFunctor* scales a *PolynomialTraits_d::Polynomial_d* p with respect to one variable, that is, it computes $b^{degree(p)} \cdot p(a/b \cdot x)$.

Note that this functor operates on the polynomial in the univariate view, that is, the polynomial is considered as a univariate homogeneous polynomial in one specific variable.

Refines

AdaptableFunctor
CopyConstructible
DefaultConstructible

Types

typedef PolynomialTraits_d::Polynomial_d *result_type;*

Operations

result_type *fo.operator()(PolynomialTraits_d::Polynomial_d p, PolynomialTraits_d::Innermost_coefficient_type a, PolynomialTraits_d::Innermost_coefficient_type b)*

Returns $b^{degree} \cdot p(a/b \cdot x)$, with respect to the outermost variable.

result_type *fo.operator()(PolynomialTraits_d::Polynomial_d p, PolynomialTraits_d::Innermost_coefficient_type a, PolynomialTraits_d::Innermost_coefficient_type b, int i)*

Same as first operator but for variable x_i.
Precondition: $0 \leq i < d$

See Also

Polynomial_d .... page 261
PolynomialTraits_d .... page 262

Concept

PolynomialTraits_d::Resultant

Definition

This *AdaptableBinaryFunction* computes the *resultant* of two polynomials f and g of type *PolynomialTraits_d::Polynomial_d* with respect to a certain variable.

Note that this functor operates on the polynomial in the univariate view, that is, the polynomial is considered as a univariate polynomial in one specific variable.

Let f and g be two univariate polynomials over some commutative ring A, where

$$f = f_m x^m + \ldots + f_0$$

and

$$g = g_n x^n + \ldots + g_0.$$

The resultant of f and g is defined as the determinant of the *Sylvester matrix*:

$$\begin{pmatrix} f_m & \cdots & f_0 & & & \\ & f_m & \cdots & f_0 & & \\ & & \ddots & & \ddots & \\ & & & f_m & \cdots & f_0 \\ g_n & \cdots & g_0 & & & \\ & g_n & \cdots & g_0 & & \\ & & \ddots & & \ddots & \\ & & & g_n & \cdots & g_0 \end{pmatrix}$$

Note that this is a $(n+m)\times(n+m)$ matrix as there are n rows for f and m rows that are used for g. The blank spaces are supposed to be filled with zeros.

advanced

Let L be the algebraic closure of A, and write f and g as

$$f := f_m \prod_{i=1}^{m}(x-\alpha_i),\ \alpha_i \in L$$

and

$$g := g_n \prod_{j=1}^{n}(x-\beta_j),\ \beta_i \in L,$$

then the resultant of f and g is (up to leading coefficients) the product of all pairwise differences of the roots of f and g, namely

$$res(f,g) = f_m^n g_n^m \prod_{i=1}^{m}\prod_{j=1}^{n}(\alpha_i - \beta_j).$$

In particular, $res(f,g) \neq 0$ iff f and g have a common factor with a positive degree in X.

advanced

There are various ways to compute the resultant. Naive options are the computation of the resultant as the determinant of the Sylvester Matrix or the Bezout Matrix as well as the so called subresultant algorithm, which is a variant of the Euclidean Algorithm. More sophisticated methods may use modular arithmetic and interpolation. For more information we refer to, e.g., [vzGG99].

Refines

AdaptableBinaryFunction
CopyConstructible
DefaultConstructible

Types

typedef PolynomialTraits_d::Coefficient_type	*result_type;*
typedef PolynomialTraits_d::Polynomial_d	*first_argument_type;*
typedef PolynomialTraits_d::Polynomial_d	*second_argument_type;*

Operations

result_type *fo(first_argument_type f, second_argument_type g)*

Computes the resultant of f and g, with respect to the outermost variable.

See Also

Polynomial_d .... page 261
PolynomialTraits_d .... page 262
PolynomialTraits_d::UnivariateContent .... page 293
PolynomialTraits_d::PolynomialSubresultants .... page 314
PolynomialTraits_d::PrincipalSubresultants .... page 318

Concept

PolynomialTraits_d::PolynomialSubresultants

Note: This functor is optional!

Definition

Computes the polynomial subresultant of two polynomials p and q of type *PolynomialTraits_d::Polynomial_d* with respect to outermost variable. Let $p = \sum_{i=0,\ldots,n} p_i x^i$ and $q = \sum_{i=0,\ldots,m} q_i x^i$, where x is the outermost variable. The i-th subresultant (with $i = 0,\ldots,\min\{n,m\}$) is defined by

$$\mathrm{Sres}_i(p,q) \quad = \quad \det \begin{pmatrix} p_n & \cdots & & \cdots & p_{2i-m+2} & x^{m-i-1}p \\ & \ddots & & & \vdots & \vdots \\ & & p_n & \cdots & p_{i+1} & p \\ q_m & \cdots & & \cdots & q_{2i-n+2} & x^{n-i-1}q \\ & \ddots & & & \vdots & \vdots \\ & & q_m & \cdots & q_{i+1} & q \end{pmatrix}$$

where p_i and q_i are set to zero if $i < 0$. In the case that $n = m$, $\mathrm{Sres_n}$ is set to q.

The result is written in an output range, starting with the 0-th subresultant $\mathrm{Sres}_0(p,q)$ (aka as the resultant of p and q).

Refines

AdaptableBinaryFunction
CopyConstructible
DefaultConstructible

Operations

template<typename OutputIterator>
OutputIterator *fo(Polynomial_d p, Polynomial_d q, OutputIterator out)*

computes the polynomial subresultants of p and q, with respect to the outermost variable. Each element is of type *PolynomialTraits_d::Polynomial_d*.

template<typename OutputIterator>
OutputIterator *fo(Polynomial_d p, Polynomial_d q, OutputIterator out, int i)*

computes the polynomial subresultants of p and q, with respect to the variable x_i.

See Also

Polynomial_d . . . . . . page 261
PolynomialTraits_d . . . . . . page 262
PolynomialTraits_d::Resultant . . . . . . page 312
PolynomialTraits_d::PrincipalSubresultants . . . . . . page 318
PolynomialTraits_d::PolynomialSubresultantsWithCofactors . . . . . . page 316
PolynomialTraits_d::SturmHabichtSequence . . . . . . page 319

PolynomialTraits_d::PolynomialSubresultantsWithCofactors

Note: This functor is optional!

Definition

Computes the polynomial subresultant of two polynomials p and q of degree n and m, respectively, as defined in the documentation of *PolynomialTraits_d::PolynomialSubresultants*. Moreover, for $\mathrm{Sres}_i(p,q)$, polynomials u_i and v_i with $\deg u_i \leq m-i-1$ and $\deg v_i \leq n-i-1$ are computed such that $\mathrm{Sres}_i(p,q) = u_i p + v_i q$. u_i and v_i are called the *cofactors* of $\mathrm{Sres}_i(p,q)$.

The result is written in three output ranges, each of length $\min\{n,m\}+1$, starting with the 0-th subresultant and the corresponding cofactors.

Refines

AdaptableBinaryFunction
CopyConstructible
DefaultConstructible

Operations

template< typename OutputIterator1, typename OutputIterator2, typename OutputIterator3 >
OutputIterator1 fo.operator()(Polynomial_d p, Polynomial_d q, OutputIterator1 sres, OutputIterator2 co_p, OutputIterator3 co_q)

computes the subresultants of p and q, and the cofactors, with respect to the outermost variable. Each element is of type *PolynomialTraits_d::Polynomial_d*.

template< typename OutputIterator1, typename OutputIterator2, typename OutputIterator3 >
OutputIterator1 fo.operator()(Polynomial_d p, Polynomial_d q, OutputIterator1 sres, OutputIterator2 co_p, OutputIterator3 co_q, int i)

computes the subresultants of p and q, and the cofactors, with respect to x_i. Each element is of type *PolynomialTraits_d::Polynomial_d*.

See Also

Polynomial_d . . . . page 261
PolynomialTraits_d . . . . page 262
PolynomialTraits_d::Resultant . . . . page 312
PolynomialTraits_d::PolynomialSubresultants . . . . page 314
PolynomialTraits_d::PrincipalSubresultants . . . . page 318
PolynomialTraits_d::SturmHabichtSequenceWithCofactors . . . . page 321

Concept

PolynomialTraits_d::PrincipalSubresultants

Note: This functor is optional!

Definition

Computes the principal subresultant of two polynomials p and q of type *PolynomialTraits_d::Coefficient_type* with respect to the outermost variable. The i-th principal subresultant, $\mathrm{sres}_i(p,q)$, is defined as the coefficient at t^i of the i-th polynomial subresultant $\mathrm{Sres}_i(p,q)$. Thus, it is either the leading coefficient of Sres_i, or zero in the case where its degree is below i.

The result is written in an output range, starting with the 0-th principal subresultant $\mathrm{sres}_0(p,q)$,aka as the resultant of p and q. (Note that $\mathrm{sres}_0(p,q) = \mathrm{Sres}_0(p,q)$ by definition)

Refines

AdaptableBinaryFunction
CopyConstructible
DefaultConstructible

Operations

template<typename OutputIterator>
OutputIterator *fo(Polynomial_d p, Polynomial_d q, OutputIterator out)*

computes the principal subresultants of p and q, with respect to the outermost variable. Each element is of type *PolynomialTraits_d::Coefficient_type*.

template<typename OutputIterator>
OutputIterator *fo(Polynomial_d p, Polynomial_d q, OutputIterator out, int i)*

computes the principal subresultants of p and q, with respect to the variable x_i.

See Also

Polynomial_d . . . page 261
PolynomialTraits_d . . . page 262
PolynomialTraits_d::Resultant . . . page 312
PolynomialTraits_d::PolynomialSubresultants . . . page 314
PolynomialTraits_d::PrincipalSturmHabichtSequence . . . page 323

PolynomialTraits_d::SturmHabichtSequence

Note: This functor is optional!

Definition

Computes the Sturm-Habicht sequence (aka the signed subresultant sequence) of a polynomial f of type *PolynomialTraits_d::Polynomial_d* with respect to a certain variable x_i. The Sturm-Habicht sequence is similar to the polynomial subresultant sequence of f and its derivative $f' := \frac{\partial f}{\partial x_i}$ with respect to x_i. The implementation is based on the following definition:

Let $n := \deg f$ and $\delta_k := (-1)^{k(k+1)/2}$. For $k \in \{0,\ldots,n\}$, the *k-th Sturm-Habicht polynomial* of f is defined as:

$$\mathrm{Stha}_k(f) = \begin{cases} f & \text{if } k = n \\ f' & \text{if } k = n-1 \\ \delta_{n-k-1}\mathrm{Sres}_k(f,f') & \text{if } 0 \le k \le n-2 \end{cases}$$

where $\mathrm{Sres}_k(f,f')$ is defined as in the concept *PolynomialTraits_d::PolynomialSubresultants*.

The result is written in an output range, starting with the 0-th Sturm-Habicht polynomial (which is equal to the discriminant of f up to a multiple of the leading coefficient).

Refines

AdaptableBinaryFunction
CopyConstructible
DefaultConstructible

Operations

template<typename OutputIterator>
OutputIterator *fo(Polynomial_d f, OutputIterator out)*

computes the Sturm-Habicht sequence of f, with respect to the outermost variable. Each element is of type *PolynomialTraits_d::Polynomial_d*.

template<typename OutputIterator>
OutputIterator *fo(Polynomial_d f, OutputIterator out, int i)*

computes the Sturm-Habicht sequence of f with respect to the variable x_i.

See Also

Polynomial_d . . . . page 261
PolynomialTraits_d . . . . page 262
PolynomialTraits_d::Resultant . . . . page 312
PolynomialTraits_d::PrincipalSturmHabichtSequence . . . . page 323
PolynomialTraits_d::SturmHabichtSequenceWithCofactors . . . . page 321
PolynomialTraits_d::PolynomialSubresultants . . . . page 314

PolynomialTraits_d::SturmHabichtSequenceWithCofactors

Concept

Note: This functor is optional!

Definition

Computes the Sturm-Habicht polynomials of a polynomial f of degree n, as defined in the documentation of *PolynomialTraits_d::SturmHabichtSequence*. Moreover, for $\mathrm{Stha}_i(f)$, polynomials u_i and v_i with $\deg u_i \leq n-i-2$ and $\deg v_i \leq n-i-1$ are computed such that $\mathrm{Sres}_i(p,q) = u_i f + v_i f'$. u_i and v_i are called the *cofactors* of $\mathrm{Stha}_i(f)$.

The result is written in three output ranges, each of length $\min\{n,m\}+1$, starting with the 0-th Sturm-Habicht polynomial $\mathrm{Stha}_0(\mathrm{f})$ and the corresponding cofactors.

Refines

AdaptableBinaryFunction
CopyConstructible
DefaultConstructible

Operations

template<typename OutputIterator1, typename OutputIterator2, typename OutputIterator3>
OutputIterator1 *fo.operator()(Polynomial_d f, OutputIterator1 stha, OutputIterator2 co_f, OutputIterator3 co_fx)*

computes the Sturm-Habicht sequence of f, and the cofactors, with respect to the outermost variable. Each element is of type *PolynomialTraits_d::Polynomial_d*.

template< typename OutputIterator1, typename OutputIterator2, typename OutputIterator3 >
OutputIterator1 *fo.operator()(Polynomial_d f, OutputIterator1 stha, OutputIterator2 co_f, OutputIterator3 co_fx, int i)*

computes the Sturm-Habicht sequence of f, and the cofactors, with respect to x_i. Each element is of type *PolynomialTraits_d::Polynomial_d*.

See Also

Polynomial_d . . . . page 261
PolynomialTraits_d . . . . page 262

PolynomialTraits_d::PrincipalSturmHabichtSequence

Concept

Note: This functor is optional!

Definition

Computes the principal leading coefficients of the Sturm-Habicht sequence of a polynomials f of type *PolynomialTraits_d::Polynomial_d* with respect a certain variable x_i. This means that for the j-th Sturm-Habicht polynomial, this methods returns the coefficient of x_i^j.

Note that the degree of the j-th Sturm-Habicht polynomial is at most j, but the principal coefficient might be zero, thus, this functor does not necessarily give the leading coefficient of the Sturm-Habicht polynomials.

In case that *PolynomialTraits_d::Coefficient_type* is *RealEmbeddable*, the function *CGAL::number_of_real_roots* can be used on the resulting sequence to count the number of distinct real roots of the polynomial f.

Refines

AdaptableBinaryFunction
CopyConstructible
DefaultConstructible

Operations

template<typename OutputIterator>
OutputIterator fo(Polynomial_d f, OutputIterator out)

computes the principal coefficients of the Sturm-Habicht sequence of f, with respect to the outermost variable. Each element is of type *PolynomialTraits_d::Coefficient_type*.

template<typename OutputIterator>
OutputIterator fo(Polynomial_d f, OutputIterator out, int i)

computes the principal coefficients of the Sturm-Habicht sequence of f with respect to the variable x_i.

See Also

Polynomial_d . . . . page 261
PolynomialTraits_d . . . . page 262
CGAL::number_of_real_roots . . . . page 377
PolynomialTraits_d::Resultant . . . . page 312
PolynomialTraits_d::SturmHabichtSequence . . . . page 319
PolynomialTraits_d::PrincipalSubresultants . . . . page 318

Class

CGAL::Polynomial<Coeff>

Definition

An instance of the data type *Polynomial* represents a polynomial $p = a_0 + a_1 * x + ...a_i * x^i$ from the ring $Coeff[x]$. *Coeff* can itself be an instance of *Polynomial*, yielding a form of multivariate polynomials.

The template argument *Coeff* must be at least a model of *IntegralDomainWithoutDivision*. For all operations naturally involving division, an *IntegralDomain* is required. *Polynomial* offers a full set of algebraic operators, i.e. binary +, −, *, / as well as +=, −=, *=, /=; not only for polynomials but also for a polynomial and a number of the coefficient type. (The / operator must only be used for integral divisions, i.e. those with remainder zero.) The operations are implemented naively: + and − need a number of *Coeff* operations which is linear in the degree while * is quadratic. Unary + and − and (in)equality ==, != are provided as well.

Polynomial is a model of *LessThanComparable* if *Coeff* is a model of *LessThanComparable*. In this case *Polynomial* provides comparison operators <, >, <=, >=, where the comparison amounts to lexicographic comparison of the coefficient sequence, with the coefficient of the highest power taking precedence over those of lower powers.

Polynomial is a model of *Fraction* if *Coeff* is a model of *Fraction*. In this case Polynomial may be decomposed into a (scalar) denominator and a compound numerator with a simpler coefficient type. Often operations can be performed faster on these denominator-free multiples.

Polynomial is a model of *Modularizable* if *Coeff* is a model of *Modularizable*, where the homomorphic map on the polynomials is simply defined as the canonical extension of the homomorphic map which is defined on the coefficient type.

Implementation

Inexact and limited-precision types can be used as coefficients, but at the user's risk. The algorithms implemented were written with exact number types in mind.

This data type is implemented as a handle type with value semantics using *CGAL::Handle_with_policy*, where *HandlePolicy* is *Handle_policy_no_union*. An important invariant to be preserved by all methods is that the coefficient sequence does not contain leading zero coefficients (where leading means at the high-degree end), with the exception that the zero polynomial is represented by a single zero coefficient.

Is Model for the Concepts

Polynomial_d	
Assignable	
CopyConstructible	
DefaultConstructible	
EqualityComparable	
ImplicitInteroperable	with int
ImplicitInteroperable	with Coeff
Fraction	if *Coeff* is model of *Fraction*
LessThanComparable	if *Coeff* is model of *LessThanComparable*
Modularizable	if *Coeff* is model of *Modularizable*.

Creation

Polynomial<Coeff> poly;	Introduces an variable initialized with 0.
Polynomial<Coeff> poly(Polynomial x);	copy constructor.
Polynomial<Coeff> poly(int i);	Constructor from int.
Polynomial<Coeff> poly(Coeff x);	Constructor from type Coeff.
template <class Forward_iterator> *Polynomial<Coeff> poly(Forward_iterator first, Forward_iterator last);*	Constructor from iterator range with value type Coeff.

Types

Operations

const_iterator	*poly.begin() const*	A const random access iterator pointing to the constant coefficient.
const_iterator	*poly.end() const*	A const random access iterator pointing beyond the leading coefficient.
int	*poly.degree() const*	The degree of the polynomial in x. The degree of the zero polynomial is 0.
NT	*poly.operator[](unsigned int i) const*	Const access to the coefficient of x^i.
NT	*poly.lcoeff() const*	Const access to the leading coefficient.
std::ostream&	*operator<<(std::ostream& os, poly)*	Writes *poly* to ostream *os*. The format depends on the *CGAL::IO::MODE* of *os*. In case the mode is *CGAL::IO::ASCII* the format is $P[d(0,a_0)(1,a_1)\dots(d,a_d)]$, where d is the degree of the polynomial. The format is output sensitive, that is, coefficients that are zero are not reported. In case the mode is *CGAL::IO::PRETTY* the format is human readable.
std::istream&	*operator>>(std::istream& is, poly)*	Reads *poly* from istream *is* in format $P[d(0,a_0)(1,a_1)\dots(d,a_d)]$, the output format in mode *CGAL::IO::ASCII*.

Class

CGAL::Polynomial_traits_d<Polynomial_d>

Definition

A model of concept *PolynomialTraits_d*

#include <CGAL/Polynomial_traits_d.h>

Is Model for the Concepts

PolynomialTraits_d

CGAL::Exponent_vector

Class

Definition

For a given (multivariate) monomial the vector of its exponents is called the exponent vector. The class is meant to represent such a vector.

A vector is considered as valid, in case it represents a valid monomial. that is, it should not contain negative exponents. We decided to use int as the value type, since negative exponents may appear in intermediate results. The set of exponent vectors with elementwise addition forms an *Abelian Group*.

Beside the constructors has almost the same interface as an *std::vector<int>*. Moreover the comparison is changed such that the lexicographic order starts the comparison at the last entry. This reflects the fact that the last entry corresponds to the outermost variable of a multivariate polynomial.

#include <CGAL/Exponent_vector.h>

Is Model for the Concepts

Random Access Container
Back Insertion Sequence

DefaultConstructible
Assignable
CopyConstructible

EqualityComparable
LessThanComparable

Creation

Exponent_vector fo;	introduces an uninitialized variable *fo*.
Exponent_vector fo(ev_);	The copy constructor
Exponent_vector fo(int e1);	Creates a vector containing the given element.
Exponent_vector fo(int e1, int e2);	Creates a vector containing the given elements.
Exponent_vector fo(int e1, int e2, int e3);	Creates a vector containing the given elements.
Exponent_vector fo(int e1, int e2, int e3, int e4);	Creates a vector containing the given elements.
template < class InputIterator > *Exponent_vector fo(InputIterator begin, InputIterator end);*	Creates a vector with a copy of the given range. *Precondition*: *InputIterator* must allow the value type *int*.

Operations

bool	*is_valid(ev)*	Returns true if all entries of ev are not negative.

Operations

Group Operation:

Exponent_vector	$+ev1$	
Exponent_vector	$-ev1$	
Exponent_vector	$ev1 + ev2$	*Precondition*: ev1.size() == ev2.size()
Exponent_vector	$ev1 - ev2$	*Precondition*: ev1.size() == ev2.size()
Exponent_vector	$fo += ev2$	*Precondition*: *fo*.size() == ev2.size()
Exponent_vector	$fo -= ev2$	*Precondition*: *fo*.size() == ev2.size()

EqualityComparable:

bool	$ev1 == ev2$
bool	$ev1\ != ev2$

LessThanComparable:

bool	$ev1 < ev2$	Lexicographic compare, starting with the *last* variable.

See Also

Polynomial_d . . . . page 261
PolynomialTraits_d . . . . page 262

CGAL::Polynomial_type_generator<T,d>

Class

Definition

This class template provides a convenient way to obtain the type representing a multivariate polynomial with d variables, where T is the innermost coefficient type. In case T happens to be a *CGAL::Polynomial* the generator will add d variables to T.

T must be a model of *IntegralDomainWithoutDivision*.
d must be of type int.

Types

Polynomial_type_generator<T,d>:: Type	The generated type.

#include <CGAL/Polynomial_type_generator.h>

See Also

CGAL::Polynomial<Coeff> . . . . page 324

Function

CGAL::get_coefficient

#include <CGAL/polynomial_utils.h>

Definition

For a given *Polynomial_d* the function *get_coefficient* adapts the according functor in *Polynomial_traits_d< Polynomial_d>*.

For more details see the concept *PolynomialTraits_d::GetCoefficient*.

template <class Polynomial_d>
Polynomial_traits_d<Polynomial_d>::get_coefficient::result_type

get_coefficient(Polynomial_d p, int i)

Adapts *Polynomial_traits_d<Polynomial_d> ::GetCoefficient*.

See Also

Polynomial_d . . . . page 261
PolynomialTraits_d . . . . page 262
PolynomialTraits_d::GetCoefficient . . . . page 269
PolynomialTraits_d::GetInnermostCoefficient . . . . page 270

CGAL::get_innermost_coefficient

#include <CGAL/polynomial_utils.h>

Definition

For a given *Polynomial_d* the function *get_innermost_coefficient* adapts the according functor in *Polynomial_traits_d<Polynomial_d>*.

For more details see the concept *PolynomialTraits_d::GetInnermostCoefficient*.

template <class Polynomial_d>
Polynomial_traits_d<Polynomial_d>::get_innermost_coefficient::result_type

get_innermost_coefficient(Polynomial_d p, Exponent_vector ev)

Adapts *Polynomial_traits_d<Polynomial_d> ::GetInnermostCoefficient*.

See Also

Polynomial_d . . . . page 261
PolynomialTraits_d . . . . page 262
PolynomialTraits_d::GetCoefficient . . . . page 269
PolynomialTraits_d::GetInnermostCoefficient . . . . page 270

Function

Function

CGAL::permute

#include <CGAL/polynomial_utils.h>

Definition

For a given *Polynomial_d* the function *permute* adapts the according functor in *Polynomial_traits_d< Polynomial_d>*.

For more details see the concept *PolynomialTraits_d::Permute*.

template <class Polynomial_d, class InputIterator >
Polynomial_traits_d<Polynomial_d>::Permute::result_type

permute(Polynomial_d p, InputIterator begin, InputIterator end)

Adapts *Polynomial_traits_d<Polynomial_d>::Permute*.

See Also

Polynomial_d . . . . page 261
PolynomialTraits_d . . . . page 262
PolynomialTraits_d::Permute . . . . page 274

CGAL::swap

#include <CGAL/polynomial_utils.h>

Definition

For a given *Polynomial_d* the function *swap* adapts the according functor in *Polynomial_traits_d<Polynomial_d>*.

For more details see the concept *PolynomialTraits_d::Swap*.

template <class Polynomial_d>
Polynomial_traits_d<Polynomial_d>::Swap::result_type

swap(Polynomial_d p, int i, int j)

Adapts *Polynomial_traits_d<Polynomial_d>::Swap*.

See Also

Polynomial_d . . . . page 261
PolynomialTraits_d . . . . page 262
PolynomialTraits_d::Swap . . . . page 275

Function

Function

CGAL::move

#include <CGAL/polynomial_utils.h>

Definition

For a given *Polynomial_d* the function *move* adapts the according functor in *Polynomial_traits_d<Polynomial_d>*.

For more details see the concept *PolynomialTraits_d::Move*.

template <class Polynomial_d>
Polynomial_traits_d<Polynomial_d>::Move::result_type

move(Polynomial_d p, int i, int j)

Adapts *Polynomial_traits_d<Polynomial_d>::Move*.

See Also

Polynomial_d . . . . page 261
PolynomialTraits_d . . . . page 262
PolynomialTraits_d::Move . . . . page 276

CGAL::degree

#include <CGAL/polynomial_utils.h>

Definition

For a given *Polynomial_d* the function *degree* adapts the according functor in *Polynomial_traits_d<Polynomial_d>*.

For more details see the concept *PolynomialTraits_d::Degree*.

template <class Polynomial_d>
Polynomial_traits_d<Polynomial_d>::Degree::result_type

degree(Polynomial_d p, int i, index = Polynomial_traits_d<Polynomial_d>::d-1)

Adapts *Polynomial_traits_d<Polynomial_d>::Degree*.

See Also

Polynomial_d . . . . page 261
PolynomialTraits_d . . . . page 262
PolynomialTraits_d::Degree . . . . page 277

Function

CGAL::total_degree

#include <CGAL/polynomial_utils.h>

Definition

For a given *Polynomial_d* the function *total_degree* adapts the according functor in *Polynomial_traits_d< Polynomial_d>*.

For more details see the concept *PolynomialTraits_d::TotalDegree*.

template <class Polynomial_d>
Polynomial_traits_d<Polynomial_d>::Total_degree::result_type

total_degree(Polynomial_d p)

Adapts *Polynomial_traits_d<Polynomial_d>::Total_degree*.

See Also

Polynomial_d . . . . page 261
PolynomialTraits_d . . . . page 262
PolynomialTraits_d::TotalDegree . . . . page 278

CGAL::degree_vector

#include <CGAL/polynomial_utils.h>

Definition

For a given *Polynomial_d* the function *degree_vector* adapts the according functor in *Polynomial_traits_d< Polynomial_d>*.

For more details see the concept *PolynomialTraits_d::DegreeVector*.

template <class Polynomial_d>
Polynomial_traits_d<Polynomial_d>::Degree_vector::result_type

degree_vector(Polynomial_d p)

Adapts *Polynomial_traits_d<Polynomial_d> ::DegreeVector*.

See Also

Polynomial_d . . . . page 261
PolynomialTraits_d . . . . page 262
PolynomialTraits_d::Degree_vector . . . . page **??**

Function

Function

CGAL::leading_coefficient

#include <CGAL/polynomial_utils.h>

Definition

For a given *Polynomial_d* the function *leading_coefficient* adapts the according functor in *Polynomial_traits_d<Polynomial_d>*.

For more details see the concept *PolynomialTraits_d::LeadingCoefficient*.

template <class Polynomial_d>
Polynomial_traits_d<Polynomial_d>::Leading_coefficient::result_type

leading_coefficient(Polynomial_d p)

Adapts *Polynomial_traits_d<Polynomial_d>::Leading_coefficient*.

See Also

Polynomial_d . . . . page 261
PolynomialTraits_d . . . . page 262
PolynomialTraits_d::LeadingCoefficient . . . . page 280

CGAL::innermost_leading_coefficient

Function

#include <CGAL/polynomial_utils.h>

Definition

For a given *Polynomial_d* the function *innermost_leading_coefficient* adapts the according functor in *Polynomial_traits_d<Polynomial_d>*.

For more details see the concept *PolynomialTraits_d::InnermostLeadingCoefficient*.

template <class Polynomial_d>
Polynomial_traits_d<Polynomial_d>::Innermost_leading_coefficient::result_type

innermost_leading_coefficient(Polynomial_d p)

Adapts *Polynomial_traits_d<Polynomial_d> ::InnermostLeadingCoefficient*.

See Also

Polynomial_d . . . . page 261
PolynomialTraits_d . . . . page 262
PolynomialTraits_d::InnermostLeadingCoefficient . . . . page 281

Function

CGAL::canonicalize

#include <CGAL/polynomial_utils.h>

Definition

For a given *Polynomial_d* the function *canonicalize* adapts the according functor in *Polynomial_traits_d< Polynomial_d>*.

For more details see the concept *PolynomialTraits_d::Canonicalize*.

template <class Polynomial_d>
Polynomial_traits_d<Polynomial_d>::Canonicalize::result_type

canonicalize(Polynomial_d p)

Adapts *Polynomial_traits_d<Polynomial_d>::Canonicalize*.

See Also

Polynomial_d . . . . page 261
PolynomialTraits_d . . . . page 262
PolynomialTraits_d::Canonicalize . . . . page 282

CGAL::differentiate

#include <CGAL/polynomial_utils.h>

Function

Definition

For a given *Polynomial_d* the function *differentiate* adapts the according functor in *Polynomial_traits_d< Polynomial_d>*.

For more details see the concept *PolynomialTraits_d::Differentiate*.

template <class Polynomial_d>
Polynomial_traits_d<Polynomial_d>::Differentiate::result_type

differentiate(Polynomial_d p, index = Polynomial_traits_d<Polynomial_d>::d-1)

Adapts *Polynomial_traits_d<Polynomial_d>::Differentiate*.

See Also

Polynomial_d . . . . page 261
PolynomialTraits_d . . . . page 262
PolynomialTraits_d::Differentiate . . . . page 283

Function

CGAL::evaluate

#include <CGAL/polynomial_utils.h>

Definition

For a given *Polynomial_d* the function *evaluate* adapts the according functor in *Polynomial_traits_d< Polynomial_d>*.

For more details see the concept *PolynomialTraits_d::Evaluate*.

template <class Polynomial_d>
Polynomial_traits_d<Polynomial_d>::Evaluate::result_type

evaluate(Polynomial_d p, Polynomial_traits_d<Polynomial_d>::Coefficient_type x)

Adapts *Polynomial_traits_d<Polynomial_d>::Evaluate*.

See Also

Polynomial_d . . . . page 261
PolynomialTraits_d . . . . page 262
PolynomialTraits_d::Evaluate . . . . page 284

CGAL::evaluate_homogeneous

#include <CGAL/polynomial_utils.h>

Definition

For a given *Polynomial_d* the function *evaluate_homogeneous* adapts the according functor in *Polynomial_traits_d<Polynomial_d>*.

For more details see the concept *PolynomialTraits_d::EvaluateHomogeneous*.

template <class Polynomial_d>
Polynomial_traits_d<Polynomial_d>::Evaluate_homogeneous::result_type

evaluate_homogeneous(Polynomial_d p,
Polynomial_traits_d<Polynomial_d>::Coefficient_type u,
Polynomial_traits_d<Polynomial_d>::Coefficient_type v)

Adapts *Polynomial_traits_d<Polynomial_d>::Evaluate_homogeneous*.

See Also

Polynomial_d . . . . page 261
PolynomialTraits_d . . . . page 262
PolynomialTraits_d::EvaluateHomogeneous . . . . page 285

Function

Function

CGAL::substitute

#include <CGAL/polynomial_utils.h>

Definition

For a given *Polynomial_d* the function *substitute* adapts the according functor in *Polynomial_traits_d< Polynomial_d>*.

For more details see the concept *PolynomialTraits_d::Substitute*.

template < class Polynomial_d, class InputIterator >
CGAL::Coercion_traits< Polynomial_traits_d<Polynomial_d>::Innermost_coefficient, std::iterator_traits< Input_iterator>::value_type >::Type

substitute(Polynomial_d p, InputIterator begin, InputIterator end)

Adapts *Polynomial_traits_d<Polynomial_d>::Substitute*.

See Also

Polynomial_d . . . . page 261
PolynomialTraits_d . . . . page 262
PolynomialTraits_d::Substitute . . . . page 286

CGAL::is_zero_at

Function

#include <CGAL/polynomial_utils.h>

Definition

For a given *Polynomial_d* the function *is_zero_at* adapts the according functor in *Polynomial_traits_d< Polynomial_d>*.

For more details see the concept *PolynomialTraits_d::IsZeroAt*.

template < class Polynomial_d, class InputIterator >
Polynomial_traits_d<Polynomial_d>::Is_zero_at::result_type

is_zero_at(Polynomial_d p, InputIterator begin, InputIterator end)

Adapts *Polynomial_traits_d<Polynomial_d>::Is_zero_at*.

See Also

Polynomial_d . . . . page 261
PolynomialTraits_d . . . . page 262
PolynomialTraits_d::IsZeroAt . . . . page 288

Function

CGAL::sign_at

#include <CGAL/polynomial_utils.h>

Definition

For a given *Polynomial_d* the function *sign_at* adapts the according functor in *Polynomial_traits_d<Polynomial_d>*.

For more details see the concept *PolynomialTraits_d::SignAt*.

template < class Polynomial_d, class InputIterator >
Polynomial_traits_d<Polynomial_d>::Sign_at::result_type

sign_at(Polynomial_d p, InputIterator begin, InputIterator end)

Adapts *Polynomial_traits_d<Polynomial_d>::Sign_at*.

See Also

Polynomial_d . . . . page 261
PolynomialTraits_d . . . . page 262
PolynomialTraits_d::SignAt . . . . page 290

CGAL::substitute_homogeneous

#include <CGAL/polynomial_utils.h>

Definition

For a given *Polynomial_d* the function *substitute_homogeneous* adapts the according functor in *Polynomial_traits_d<Polynomial_d>*.

For more details see the concept *PolynomialTraits_d::SubstituteHomogeneous*.

template < class Polynomial_d, class InputIterator >
CGAL::Coercion_traits< Polynomial_traits_d<Polynomial_d>::Innermost_coefficient, std::iterator_traits< Input_iterator>::value_type >::Type

substitute_homogeneous(Polynomial_d p, InputIterator begin, InputIterator end)

Adapts *Polynomial_traits_d<Polynomial_d>::Substitute_homogeneous*.

See Also

Polynomial_d . . . . page 261
PolynomialTraits_d . . . . page 262
PolynomialTraits_d::SubstituteHomogeneous . . . . page 287

Function

CGAL::is_zero_at_homogeneous

#include <CGAL/polynomial_utils.h>

Definition

For a given *Polynomial_d* the function *is_zero_at_homogeneous* adapts the according functor in *Polynomial_traits_d<Polynomial_d>*.

For more details see the concept *PolynomialTraits_d::IsZeroAtHomogeneous*.

template < class Polynomial_d, class InputIterator >
Polynomial_traits_d<Polynomial_d>::Is_zero_at_homogeneous::result_type

is_zero_at_homogeneous(Polynomial_d p, InputIterator begin, InputIterator end)

Adapts *Polynomial_traits_d<Polynomial_d>::Is_zero_at_homogeneous*.

See Also

Polynomial_d . . . page 261
PolynomialTraits_d . . . page 262
PolynomialTraits_d::IsZeroAtHomogeneous . . . page 289

CGAL::sign_at_homogeneous

#include <CGAL/polynomial_utils.h>

Definition

For a given *Polynomial_d* the function *sign_at_homogeneous* adapts the according functor in *Polynomial_traits_d<Polynomial_d>*.

For more details see the concept *PolynomialTraits_d::SignAtHomogeneous*.

template < class Polynomial_d, class InputIterator >
Polynomial_traits_d<Polynomial_d>::Sign_at_homogeneous::result_type

sign_at_homogeneous(Polynomial_d p, InputIterator begin, InputIterator end)

Adapts *Polynomial_traits_d<Polynomial_d>::Sign_at_homogeneous*.

See Also

CGAL::compare

#include <CGAL/polynomial_utils.h>

Definition

For a given *Polynomial_d* the function *compare* adapts the according functor in *Polynomial_traits_d< Polynomial_d>*.

For more details see the concept *PolynomialTraits_d::Compare*.

template <class Polynomial_d>
Polynomial_traits_d<Polynomial_d>::Compare::result_type

compare(Polynomial_d p, Polynomial_d q)

Adapts *Polynomial_traits_d<Polynomial_d>::Compare*.

See Also

Polynomial_d . . . . page 261
PolynomialTraits_d . . . . page 262
PolynomialTraits_d::Compare . . . . page 292

CGAL::univariate_content

#include <CGAL/polynomial_utils.h>

Function

Definition

For a given *Polynomial_d* the function *univariate_content* adapts the according functor in *Polynomial_traits_d< Polynomial_d>*.

For more details see the concept *PolynomialTraits_d::UnivariateContent*.

template <class Polynomial_d>
Polynomial_traits_d<Polynomial_d>::Univariate_content::result_type

univariate_content(Polynomial_d p)

Adapts *Polynomial_traits_d<Polynomial_d> ::UnivariateContent*.

See Also

Polynomial_d . . . . page 261
PolynomialTraits_d . . . . page 262
PolynomialTraits_d::Univariate_Content . . . . page **??**

Function

CGAL::multivariate_content

#include <CGAL/polynomial_utils.h>

Definition

For a given *Polynomial_d* the function *multivariate_content* adapts the according functor in *Polynomial_traits_d<Polynomial_d>*.

For more details see the concept *PolynomialTraits_d::MultivariateContent*.

template <class Polynomial_d>
Polynomial_traits_d<Polynomial_d>::Multivariate_content::result_type

multivariate_content(Polynomial_d p)

Adapts *Polynomial_traits_d<Polynomial_d>::Multivariate_content*.

See Also

Polynomial_d . . . . page 261
PolynomialTraits_d . . . . page 262
PolynomialTraits_d::MultivariateContent . . . . page 294

CGAL::square_free_factorize

#include <CGAL/polynomial_utils.h>

Definition

For a given *Polynomial_d* the function *square_free_factorize* adapts the according functor in *Polynomial_traits_d<Polynomial_d>*.

For more details see the concept *PolynomialTraits_d::SquareFreeFactorize*.

template <class Polynomial_d, class OutputIterator >
OutputIterator square_free_factorize(Polynomial_d p, OutputIterator it, Polynomial_traits_d<Polynomial>::Innermost_coefficient& a)

Adapts *Polynomial_traits_d<Polynomial_d>::Square_free_factorize*.

template <class Polynomial_d, class OutputIterator >
OutputIterator square_free_factorize(Polynomial_d p, OutputIterator it)

Adapts *Polynomial_traits_d<Polynomial_d>::Square_free_factorize*.

See Also

Polynomial_d . . . . page 261
PolynomialTraits_d . . . . page 262
PolynomialTraits_d::SquareFreeFactorize . . . . page 295

Function

Function

CGAL::make_square_free

#include <CGAL/polynomial_utils.h>

Definition

For a given *Polynomial_d* the function *make_square_free* adapts the according functor in *Polynomial_traits_d< Polynomial_d>*.

For more details see the concept *PolynomialTraits_d::MakeSquareFree*.

template <class Polynomial_d>
Polynomial_traits_d<Polynomial_d>::Make_square_free::result_type

make_square_free(Polynomial_d p)

Adapts *Polynomial_traits_d<Polynomial_d>::Make_square_free*.

See Also

Polynomial_d . . . . page 261
PolynomialTraits_d . . . . page 262
PolynomialTraits_d::MakeSquareFree . . . . page 296

CGAL::is_square_free

#include <CGAL/polynomial_utils.h>

Definition

For a given *Polynomial_d* the function *is_square_free* adapts the according functor in *Polynomial_traits_d< Polynomial_d>*.

For more details see the concept *PolynomialTraits_d::IsSquareFree*.

template <class Polynomial_d>
Polynomial_traits_d<Polynomial_d>::Is_square_free::result_type

is_square_free(Polynomial_d p)

Adapts *Polynomial_traits_d<Polynomial_d>::Is_square_free*.

See Also

Polynomial_d . . . . page 261
PolynomialTraits_d . . . . page 262
PolynomialTraits_d::IsSquareFree . . . . page 297

Function

Function

CGAL::pseudo_division

#include <CGAL/polynomial_utils.h>

Definition

For a given *Polynomial_d* the function *pseudo_division* adapts the according functor in *Polynomial_traits_d< Polynomial_d>*.

For more details see the concept *PolynomialTraits_d::PseudoDivision*.

template <class Polynomial_d>
void *pseudo_division(Polynomial_d f, Polynomial_d g, Polynomial_d& q, Polynomial_d& r, Polynomial_traits_d<Polynomial_d>::Coefficient_type& D)*

Adapts *Polynomial_traits_d<Polynomial_d>::Pseudo_division*.

See Also

CGAL::pseudo_division_quotient

#include <CGAL/polynomial_utils.h>

Definition

For a given *Polynomial_d* the function *pseudo_division_quotient* adapts the according functor in *Polynomial_traits_d<Polynomial_d>*.

For more details see the concept *PolynomialTraits_d::PseudoDivisionQuotient*.

template <class Polynomial_d>
Polynomial_traits_d<Polynomial_d>::Pseudo_division_quotient::result_type

pseudo_division_quotient(Polynomial_d p, Polynomial_d q)

Adapts *Polynomial_traits_d<Polynomial_d>::Pseudo_division_quotient*.

See Also

Polynomial_d . . . . page 261
PolynomialTraits_d . . . . page 262
PolynomialTraits_d::PseudoDivisionQuotient . . . . page 299

Function

Function

CGAL::pseudo_division_remainder

#include <CGAL/polynomial_utils.h>

Definition

For a given *Polynomial_d* the function *pseudo_division_remainder* adapts the according functor in *Polynomial_traits_d<Polynomial_d>*.

For more details see the concept *PolynomialTraits_d::PseudoDivisionRemainder*.

template <class Polynomial_d>
Polynomial_traits_d<Polynomial_d>::Pseudo_division_remainder::result_type

pseudo_division_remainder(Polynomial_d p, Polynomial_d q)

Adapts *Polynomial_traits_d<Polynomial_d>::Pseudo_division_remainder*.

See Also

Polynomial_d . . . . page 261
PolynomialTraits_d . . . . page 262
PolynomialTraits_d::PseudoDivisionRemainder . . . . page 300

CGAL::gcd_up_to_constant_factor

Function

#include <CGAL/polynomial_utils.h>

Definition

For a given *Polynomial_d* the function *gcd_up_to_constant_factor* adapts the according functor in *Polynomial_traits_d<Polynomial_d>*.

For more details see the concept *PolynomialTraits_d::GcdUpToConstantFactor*.

template <class Polynomial_d>
Polynomial_traits_d<Polynomial_d>::Gcd_up_to_constant_factor::result_type

gcd_up_to_constant_factor(Polynomial_d p, Polynomial_d q)

Adapts *Polynomial_traits_d<Polynomial_d>::Gcd_up_to_constant_factor*.

See Also

Polynomial_d . . . . page 261
PolynomialTraits_d . . . . page 262
PolynomialTraits_d::GcdUpToConstantFactor . . . . page 301

Function

CGAL::integral_division_up_to_constant_factor

#include <CGAL/polynomial_utils.h>

Definition

For a given *Polynomial_d* the function *integral_division_up_to_constant_factor* adapts the according functor in *Polynomial_traits_d<Polynomial_d>*.

For more details see the concept *PolynomialTraits_d::IntegralDivisionUpToConstantFactor*.

template <class Polynomial_d>
Polynomial_traits_d<Polynomial_d>::Integral_division_up_to_constant_factor::result_type

integral_division_up_to_constant_factor(Polynomial_d p, Polynomial_d q)

Adapts *Polynomial_traits_d<Polynomial_d>::Integral_division_up_to_constant_factor*.

See Also

CGAL::univariate_content_up_to_constant_factor

Function

#include <CGAL/polynomial_utils.h>

Definition

For a given *Polynomial_d* the function *univariate_content_up_to_constant_factor* adapts the according functor in *Polynomial_traits_d<Polynomial_d>*.

For more details see the concept *PolynomialTraits_d::UnivariateContentUpToConstantFactor*.

template <class Polynomial_d>
Polynomial_traits_d<Polynomial_d>::Univariate_content_up_to_constant_factor::result_type

univariate_content_up_to_constant_factor(Polynomial_d p)

Adapts *Polynomial_traits_d<Polynomial_d>::Univariate_content_up_to_constant_factor*.

See Also

Polynomial_d . . . . page 261
PolynomialTraits_d . . . . page 262
PolynomialTraits_d::UnivariateContentUpToConstantFactor . . . . page 303

Function

CGAL::square_free_factorize_up_to_constant_factor

#include <CGAL/polynomial_utils.h>

Definition

For a given *Polynomial_d* the function *square_free_factorize_up_to_constant_factor* adapts the according functor in *Polynomial_traits_d<Polynomial_d>*.

For more details see the concept *PolynomialTraits_d::SquareFreeFactorizeUpToConstantFactor*.

template <class Polynomial_d, class OutputIterator >
OutputIterator *square_free_factorize_up_to_constant_factor(Polynomial_d p, OutputIterator it)*

Adapts *Polynomial_traits_d<Polynomial_d>::Square_free_factorize_up_to_constant_factor*.

See Also

Polynomial_d . . . . page 261
PolynomialTraits_d . . . . page 262
PolynomialTraits_d::SquareFreeFactorizeUpToConstantFactor . . . . page 304

CGAL::shift

#include <CGAL/polynomial_utils.h>

Definition

For a given *Polynomial_d* the function *shift* adapts the according functor in *Polynomial_traits_d<Polynomial_d>*.

For more details see the concept *PolynomialTraits_d::Shift*.

template <class Polynomial_d>
Polynomial_traits_d<Polynomial_d>::Shift::result_type

shift(Polynomial_d p, int i, int index = Polynomial_traits_d<Polynomial_d>::d-1)

Adapts *Polynomial_traits_d<Polynomial_d>::Shift*.

See Also

Polynomial_d . . . . page 261
PolynomialTraits_d . . . . page 262
PolynomialTraits_d::Shift . . . . page 305

Function

CGAL::negate

#include <CGAL/polynomial_utils.h>

Definition

For a given *Polynomial_d* the function *negate* adapts the according functor in *Polynomial_traits_d<Polynomial_d>*.

For more details see the concept *PolynomialTraits_d::Negate*.

template <class Polynomial_d>
Polynomial_traits_d<Polynomial_d>::Negate::result_type

negate(Polynomial_d p, int index = Polynomial_traits_d<Polynomial_d>::d-1)

Adapts *Polynomial_traits_d<Polynomial_d>::Negate*.

See Also

Polynomial_d . . . . page 261
PolynomialTraits_d . . . . page 262
PolynomialTraits_d::Negate . . . . page 306

CGAL::invert

#include <CGAL/polynomial_utils.h>

Definition

For a given *Polynomial_d* the function *invert* adapts the according functor in *Polynomial_traits_d<Polynomial_d>*.

For more details see the concept *PolynomialTraits_d::Invert*.

template <class Polynomial_d>
Polynomial_traits_d<Polynomial_d>::Invert::result_type

invert(Polynomial_d p, int index = Polynomial_traits_d<Polynomial_d>::d-1)

Adapts *Polynomial_traits_d<Polynomial_d>::Invert*.

See Also

Polynomial_d . . . . page 261
PolynomialTraits_d . . . . page 262
PolynomialTraits_d::Invert . . . . page 307

Function

Function

CGAL::translate

#include <CGAL/polynomial_utils.h>

Definition

For a given *Polynomial_d* the function *translate* adapts the according functor in *Polynomial_traits_d< Polynomial_d>*.

For more details see the concept *PolynomialTraits_d::Translate*.

template <class Polynomial_d>
Polynomial_traits_d<Polynomial_d>::Translate::result_type

translate(Polynomial_d p,
Polynomial_traits_d<Polynomial_d>::Innermost_coefficient_type a,
int index = Polynomial_traits_d<Polynomial_d>::d-1)

Adapts *Polynomial_traits_d<Polynomial_d>::Translate*.

See Also

Polynomial_d . . . . page 261
PolynomialTraits_d . . . . page 262
PolynomialTraits_d::Translate . . . . page 308

CGAL::translate_homogeneous

#include <CGAL/polynomial_utils.h>

Definition

For a given *Polynomial_d* the function *translate_homogeneous* adapts the according functor in *Polynomial_traits_d<Polynomial_d>*.

For more details see the concept *PolynomialTraits_d::TranslateHomogeneous*.

template <class Polynomial_d>
Polynomial_traits_d<Polynomial_d>::Translate_homogeneous::result_type

translate_homogeneous(Polynomial_d p,
Polynomial_traits_d<Polynomial_d>::Innermost_coefficient_type u,
Polynomial_traits_d<Polynomial_d>::Innermost_coefficient_type v,
int index = Polynomial_traits_d<Polynomial_d>::d-1)

Adapts *Polynomial_traits_d<Polynomial_d>::Translate_homogeneous*.

See Also

Polynomial_d . . . . . page 261
PolynomialTraits_d . . . . . page 262
PolynomialTraits_d::TranslateHomogeneous . . . . . page 309

Function

CGAL::scale

#include <CGAL/polynomial_utils.h>

Definition

For a given *Polynomial_d* the function *scale* adapts the according functor in *Polynomial_traits_d<Polynomial_d>*.

For more details see the concept *PolynomialTraits_d::Scale*.

template <class Polynomial_d>
Polynomial_traits_d<Polynomial_d>::Scale::result_type

scale(Polynomial_d p,
Polynomial_traits_d<Polynomial_d>::Innermost_coefficient_type a,
int index = Polynomial_traits_d<Polynomial_d>::d-1)

Adapts *Polynomial_traits_d<Polynomial_d>::Scale*.

See Also

Polynomial_d . . . . page 261
PolynomialTraits_d . . . . page 262
PolynomialTraits_d::Scale . . . . page 310

CGAL::scale_homogeneous

#include <CGAL/polynomial_utils.h>

Definition

For a given *Polynomial_d* the function *scale_homogeneous* adapts the according functor in *Polynomial_traits_d<Polynomial_d>*.

For more details see the concept *PolynomialTraits_d::ScaleHomogeneous*.

template <class Polynomial_d>
Polynomial_traits_d<Polynomial_d>::Scale_homogeneous::result_type

scale_homogeneous(Polynomial_d p,
Polynomial_traits_d<Polynomial_d>::Innermost_coefficient_type u,
Polynomial_traits_d<Polynomial_d>::Innermost_coefficient_type v,
int index = Polynomial_traits_d<Polynomial_d>::d-1)

Adapts *Polynomial_traits_d<Polynomial_d>::Scale_homogeneous*.

See Also

Polynomial_d . . . . . . page 261
PolynomialTraits_d . . . . . . page 262
PolynomialTraits_d::ScaleHomogeneous . . . . . . page 311

Function

CGAL::resultant

#include <CGAL/polynomial_utils.h>

Definition

For a given *Polynomial_d* the function *resultant* adapts the according functor in *Polynomial_traits_d< Polynomial_d>*.

For more details see the concept *PolynomialTraits_d::Resultant*.

template <class Polynomial_d>
Polynomial_traits_d<Polynomial_d>::Resultant::result_type

resultant(Polynomial_d p, Polynomial_d q)

Adapts *Polynomial_traits_d<Polynomial_d>::Resultant*.

See Also

Polynomial_d . . . . page 261
PolynomialTraits_d . . . . page 262
PolynomialTraits_d::Resultant . . . . page 312

CGAL::polynomial_subresultants

#include <CGAL/polynomial_utils.h>

Definition

For a given *Polynomial_d* the function *polynomial_subresultants* adapts the according functor in *Polynomial_traits_d<Polynomial_d>*.

For more details see the concept *PolynomialTraits_d::PolynomialSubresultants*.

template<typename Polynomial_d,typename OutputIterator>
OutputIterator *polynomial_subresultants(Polynomial_d p, Polynomial_d q, OutputIterator out)*

computes the polynomial subresultants of p and q, with respect to the outermost variable. Each element is of type *Polynomial_d*.

See Also

Polynomial_d . . . . page 261
PolynomialTraits_d . . . . page 262
PolynomialTraits_d::PolynomialSubresultants . . . . page 314

Function

Function

CGAL::polynomial_subresultants_with_cofactors

#include <CGAL/polynomial_utils.h>

Definition

For a given *Polynomial_d* the function *polynomial_subresultants_with_cofactors* adapts the according functor in *Polynomial_traits_d<Polynomial_d>*.

For more details see the concept *PolynomialTraits_d::PolynomialSubresultantsWithCofactors*.

template<typename Polynomial_d, typename OutputIterator1, typename OutputIterator2, typename OutputIterator3>
OutputIterator1 polynomial_subresultants_with_cofactors(Polynomial_d p, Polynomial_d q, OutputIterator1 sres_out, OutputIterator2 coP_out, OutputIterator3 coQ_out)

computes the polynomial subresultants of p and q, *sres_out*, with respect to the outermost variable, and the cofactors for P, *coP_out* and Q, *coQ_out*. The elements of each output range are of type *Polynomial_d*.

See Also

Polynomial_d . . . page 261
PolynomialTraits_d . . . page 262
PolynomialTraits_d::PolynomialSubresultantsWithCofactors . . . page 316

CGAL::principal_subresultants

#include <CGAL/polynomial_utils.h>

Definition

For a given *Polynomial_d* the function *principal_subresultants* adapts the according functor in *Polynomial_traits_d<Polynomial_d>*.

For more details see the concept *PolynomialTraits_d::PrincipalSubresultants*.

template<typename Polynomial_d,typename OutputIterator>
OutputIterator *principal_subresultants(Polynomial_d p, Polynomial_d q, OutputIterator out)*

computes the principal subresultants of p and q, with respect to the outermost variable. Each element is of type *Polynomial_traits_d<Polynomial_d>::Coefficient_type*.

See Also

Polynomial_d . . . . page 261
PolynomialTraits_d . . . . page 262
PolynomialTraits_d::PrincipalSubresultants . . . . page 318

Function

Function

CGAL::sturm_habicht_sequence

#include <CGAL/polynomial_utils.h>

Definition

For a given *Polynomial_d* the function *sturm_habicht_sequence* adapts the according functor in *Polynomial_traits_d<Polynomial_d>*.

For more details see the concept *PolynomialTraits_d::SturmHabichtSequence*.

template<typename Polynomial_d,typename OutputIterator>
OutputIterator *sturm_habicht_sequence(Polynomial_d f, OutputIterator out)*

computes the Sturm-Habicht-sequence of f with respect to the outermost variable. Each element is of type *Polynomial_d*.

See Also

Polynomial_d . . . . page 261
PolynomialTraits_d . . . . page 262
PolynomialTraits_d::SturmHabichtSequence . . . . page 319

CGAL::sturm_habicht_sequence_with_cofactors

#include <CGAL/polynomial_utils.h>

Definition

For a given *Polynomial_d* the function *sturm_habicht_sequence_with_cofactors* adapts the according functor in *Polynomial_traits_d<Polynomial_d>*.

For more details see the concept *PolynomialTraits_d::SturmHabichtSequenceWithCofactors*.

template<typename Polynomial_d, typename OutputIterator1, typename OutputIterator2, typename OutputIterator3>
OutputIterator1 sturm_habicht_sequence_with_cofactors(Polynomial_d f, OutputIterator1 stha_out, OutputIterator2 cof_out, OutputIterator3 cofx_out)

computes the Sturm-Habicht sequence of f *stha_out*, with respect to the outermost variable, and the cofactors for f, *cof_out* and f', *cofx_out*. The elements of each output range are of type *Polynomial_d*.

See Also

Polynomial_d . . . . page 261
PolynomialTraits_d . . . . page 262
PolynomialTraits_d::SturmHabichtSequenceWithCofactors . . . . page 321

Function

CGAL::principal_sturm_habicht_sequence

#include <CGAL/polynomial_utils.h>

Definition

For a given *Polynomial_d* the function *principal_sturm_habicht_sequence* adapts the according functor in *Polynomial_traits_d<Polynomial_d>*.

For more details see the concept *PolynomialTraits_d::PrincipalSturmHabichtSequence*.

template <typename Polynomial_d,typename OutputIterator>
inline OutputIterator *principal_sturm_habicht_sequence(typename Polynomial_d f, OutputIterator out)*

computes the principal Sturm-Habicht coefficients of f with respect to the outermost variable. Each element is of type *Polynomial_traits_d<Polynomial_d>::Coefficient_type*.

See Also

Polynomial_d . . . . page 261
PolynomialTraits_d . . . . page 262
PolynomialTraits_d::PrincipalSturmHabichtSequence . . . . page 323

CGAL::number_of_real_roots

#include <CGAL/polynomial_utils.h>

Definition

Given a polynomial f, or a range of values that is interpreted as the principal Sturm-Habicht coefficients of f, the function computes

$$m := \#\{\alpha \in \mathbb{R} \mid f(\alpha) = 0\}$$

that is, the number of distinct real roots of f.

The coefficient type of the polynomial, or the value type of the iterator range, respectively must be a model of *RealEmbeddable*. In the second version, it is not required to pass the exact principal Sturm-Habicht coefficients to the functions; it is only required that the sign of each element corresponds to the sign of the actual principal Sturm-Habicht coefficient.

advanced

We explain the internals of this function. For a sequence $I := (a_0, \ldots, a_n)$ of real numbers with $a_0 \neq 0$, define

$$C(I) = \sum_{i=1}^{s} \varepsilon_i$$

where s is the number of subsequences of I of the form

$$(a, \underbrace{0, \ldots, 0}_{k}, b)$$

with $a \neq 0, b \neq 0, k \geq 0$.
For the i-th subsequence of I, define

$$\varepsilon_i := \begin{array}{cl} 0 & \text{if } k \text{ is odd,} \\ (-1)^{k/2}\operatorname{sign}(ab) & \text{if } k \text{ is even.} \end{array}$$

For $f \in \mathbb{R}[x]$ with $\deg f = n$, we have:

$$C(\mathrm{stha}_n(f), \ldots, \mathrm{stha}_0(f)) = \#\{\alpha \in \mathbb{R} \mid f(\alpha) = 0\}$$

In other words, the signs of the principal Sturm-Habicht coefficients determine the number of distinct real roots of f.

advanced

Operations

template<typename Polynomial_d>
int *number_of_real_roots(Polynomial_d f)*

computes the number of distinct real roots of f

template<typename InputIterator>

int *number_of_real_roots(InputIterator start, InputIterator end)*

computes the number of distinct real roots of f whose principal Sturm-Habicht coefficients are passed by the iterator range.

See Also

Polynomial_d . . . . page 261
PolynomialTraits_d . . . . page 262
PolynomialTraits_d::PrincipalSturmHabichtSequence . . . . page 323

Chapter 8

Algebraic Kernel

Eric Berberich, Michael Hemmer, Michael Kerber, Sylvain Lazard, Luis Peñaranda, and Monique Teillaud

Contents

8.1 Introduction

Real solving of polynomials is a fundamental problem with a wide application range. This package is targeted at providing black-box implementations of state-of-the-art algorithms to determine, compare, and approximate real roots of univariate polynomials and bivariate polynomial systems. Such a black-box is called an **Algebraic Kernel**. Since this package is aimed at providing more than one implementation, the interface of the algebraic kernels is expressed in concepts. The main concepts provided by this package are the *AlgebraicKernel_d_1* for univariate polynomial systems and *AlgebraicKernel_d_2* for bivariate polynomial systems, the latter being a refinement of the first.

8.2 Algebraic Kernel Concepts

8.2.1 Univariate Algebraic Kernel

Major types

First of all, the univariate algebraic kernel provides construction, comparison and approximation of real roots of univariate polynomials. Thus, the major public types the *AlgebraicKernel_d_1* provides are:
AlgebraicKernel_d_1::Polynomial_1 – the type representing univariate polynomials,
AlgebraicKernel_d_1::Coefficient – the coefficient type of these polynomials,
AlgebraicKernel_d_1::Algebraic_real_1 – the type representing real roots,
AlgebraicKernel_d_1::Bound – the type which is used to approximate these algebraic reals, in particular, it is used to represent the boundaries of isolating intervals.

Construction of Algebraic Real Numbers

The kernel provides two different function objects to construct an *AlgebraicKernel_d_1::Algebraic_real_1*. The most general way is to use *AlgebraicKernel_d_1::Isolate_1*; The function object takes a univariate polynomial and writes all real roots into a given output iterator. It is also possible to retrieve the multiplicity of each root. The second option is to construct one particular algebraic real using *AlgebraicKernel_d_1::Construct_algebraic_real_1*. This function object provides construction from the native int type, the coefficient type as well as the bound type. Moreover, it is possible to construct an algebraic real by giving a polynomial and either an isolating interval or the index of the root. A related function object is *AlgebraicKernel_d_1::Number_of_solutions_1* computing the number of real roots of a polynomial.

Comparison and Approximation of Algebraic Real Numbers

An *AlgebraicKernel_d_1::Algebraic_real_1* is model of *RealEmbeddable*, for instance, it is possible to compare two algebraic reals, to determine the sign of an algebraic real or to ask for its double approximation, see also section 4.3. Moreover, *AlgebraicKernel_d_1::Compare_1* provides comparison with int, the coefficient type and the bound type.

There are several ways to approximate an *AlgebraicKernel_d_1::Algebraic_real_1*:
AlgebraicKernel_d_1::Approximate_absolute_1 – provides an approximation that is better than the passed absolute error bound,
AlgebraicKernel_d_1::Approximate_relative_1 – provides an approximation that is better than the passed relative error bound,
AlgebraicKernel_d_1::Isolate_1 – returns an isolating interval with respect to a given univariate polynomial,
A related function object is *AlgebraicKernel_d_1::Bound_between_1*, which computes a number that isolates two algebraic real numbers.

Interplay with Polynomials

It is also possible to retrieve a representing polynomial from an algebraic real using *AlgebraicKernel_d_1::Compute_polynomial_1*, which guarantees that the algebraic real is a root of the returned polynomial. As the name already indicates, this operation may be very costly since the polynomial may not be computed yet. Moreover, it is not guaranteed that the returned polynomial is the minimal polynomial of the number. Together with *AlgebraicKernel_d_1::Isolate_1*, it is possible to retrieve the traditional representation of an algebraic real as a square free polynomial and an isolating interval.

Though the *AlgebraicKernel_d_1* does not provide arithmetic operations on *AlgebraicKernel_d_1::Algebraic_real_1*, it is possible to compute the sign of a polynomial at a given algebraic real using *AlgebraicKernel_d_1::Sign_at_1*. Or alternatively, just compute whether the polynomial is zero at an algebraic real number using *AlgebraicKernel_d_1::Is_zero_at_1*. Note that this operation can be significantly less expensive, in particular if the polynomial is not zero at the given algebraic real.

Auxiliary Functionality for Polynomials

First of all the type *AlgebraicKernel_d_1::Polynomial_1* is required to be a model of the concept *Polynomial_1*, which is defined in the Polynomial package (see chapter 7). This implies that all essential functionality is provided via *CGAL::Polynomial_traits_d*. However, the algebraic kernel also provides several function objects to handle polynomials:
AlgebraicKernel_d_1::Is_square_free_1 – determines whether a polynomial is square free,
AlgebraicKernel_d_1::Make_square_free_1 – computes the square free part of a polynomial,
AlgebraicKernel_d_1::Square_free_factorize_1 – computes a square free factorization of a polynomial,
AlgebraicKernel_d_1::Is_coprime_1 – computes whether a pair of polynomials is square free,
AlgebraicKernel_d_1::Make_coprime_1 – decomposes two polynomials into the coprime factors and their common factor.

Though the polynomial package provides similar functionality we suggest to use the function objects provided by the kernel, since the design of the algebraic kernel allows for instance internal caching by the kernel.

Also note that *AlgebraicKernel_d_1::Square_free_factorize_1* only computes the square free factorization up to a constant factor. This is a slight modification with respect to its counterpart in *CGAL::Polynomial_traits_d*. In this way it was possible that the concepts just require the coefficient type to be a model of *IntegralDomain*, instead of *Field* or *UniqueFactorizationDomain*. For more details see also:
PolynomialTraits_d::SquareFreeFactorize .... page 295
PolynomialTraits_d::SquareFreeFactorizeUpToConstantFactor .... page 304

Design Rationale

Most implementations of an *AlgebraicKernel_d_1* will represent an algebraic real number by the root of a square free polynomial and an isolating interval, that is, the number is defined as the only root of the polynomial within the interval. Usually, one will refrain from computing the minimal polynomial since the computation of the minimal polynomial is much more expensive and does not pay of. However, besides the representation by a polynomial and an isolating interval one can also imagine the representation by a polynomial and the index of the root, e.g., as the ith real root when enumerated from minus to plus infinity. Moreover, it may very well be that the kernel just computes an approximation of the number, whereas the representing polynomial is not computed yet. This is in particular relevant in relation to the *AlgebraicKernel_d_2*, where *AlgebraicKernel_d_1::Algebraic_real_1* is used to represent coordinates of solutions of bivariate systems. Hence, the design does not allow a direct access to any, seemingly obvious, members of an *AlgebraicKernel_d_1::Algebraic_real_1*. Instead there is, e.g., *AlgebraicKernel_d_1::Compute_polynomial_1* which emphasizes that the requested polynomial may not be computed yet. Similarly, there is no way to directly ask for the refinement of the current isolating interval since this would impose a state to every object of an *AlgebraicKernel_d_1::Algebraic_real_1*.

8.2.2 Bivariate Algebraic Kernel

The concept *AlgebraicKernel_d_2* is a refinement of *AlgebraicKernel_d_1*, that is, a model of *AlgebraicKernel_d_2* is also a model of *AlgebraicKernel_d_1*. Hence, the *AlgebraicKernel_d_2* concept is designed such that occurring names and functionalities are as similar as possible to those in the *AlgebraicKernel_d_1* concept. The following are a direct generalization of their univariate counterparts:

AlgebraicKernel_d_2::Polynomial_2,
AlgebraicKernel_d_2::Algebraic_real_2,
AlgebraicKernel_d_2::Construct_algebraic_real_2,
AlgebraicKernel_d_2::Isolate_2,
AlgebraicKernel_d_2::Is_square_free_2,
AlgebraicKernel_d_2::Make_square_free_2,
AlgebraicKernel_d_2::Square_free_factorize_2,
AlgebraicKernel_d_2::Is_coprime_2,
AlgebraicKernel_d_2::Make_coprime_2,
AlgebraicKernel_d_2::Solve_2,
AlgebraicKernel_d_2::Number_of_solutions_2,
AlgebraicKernel_d_2::Compare_xy_2,
AlgebraicKernel_d_2::Sign_at_2,
AlgebraicKernel_d_2::Is_zero_at_2.

For instance, *AlgebraicKernel_d_2::Solve_2* provides the solution for a bivariate polynomial system. However, it is also possible to obtain the coordinates of these solutions with the additional functors:

AlgebraicKernel_d_2::Compute_x_2,
AlgebraicKernel_d_2::Compute_y_2.

In principal this would be sufficient generalization, since functions such as isolating, approximating algebraic real numbers could be implemented using these access functions ant the corresponding functionalities in the univariate algebraic kernel. However, one should be aware that an *AlgebraicKernel_d_2::Algebraic_real_2* is not necessarily represented as a pair of univariate solutions, that is, using *AlgebraicKernel_d_2::Compute_y_2* may entail considerable computations. Therefore, the concept also requires the following additional functors that may allow a model to bypass this issue:

AlgebraicKernel_d_2::Compute_polynomial_x_2,
AlgebraicKernel_d_2::Compute_polynomial_y_2,
AlgebraicKernel_d_2::Isolate_x_2,
AlgebraicKernel_d_2::Isolate_y_2,
AlgebraicKernel_d_2::Compare_x_2,
AlgebraicKernel_d_2::Compare_y_2,
AlgebraicKernel_d_2::Approximate_absolute_x_2,
AlgebraicKernel_d_2::Approximate_relative_x_2,
AlgebraicKernel_d_2::Approximate_absolute_y_2,
AlgebraicKernel_d_2::Approximate_relative_y_2,
AlgebraicKernel_d_2::Bound_between_x_2,
AlgebraicKernel_d_2::Bound_between_y_2.

8.3 Models

8.3.1 Generic Algebraic Kernels

The package provides generic models of the univariate and bivariate algebraic kernel, namely *CGAL::Algebraic_kernel_d_1<Coeff>* and *CGAL::Algebraic_kernel_d_2<Coeff>*, respectively. Both kernels support a large set of number types as their template argument, which defines the supported coefficient type. The supported types are, for instance, *Gmpz* and *Gmpq* as well as the corresponding types of LEDA and CORE.

The *CGAL::Algebraic_kernel_d_1<Coeff>* represents an algebraic real root by a square free polynomial and an isolating interval that uniquely defines the root. The current method to isolate roots is the Bitstream Descartes method [Eig08]. The used method to refine the approximation of an algebraic real root is a slightly modified (filtered) version of the one presented in [Abb]. The method has quadratic convergence.

CGAL::Algebraic_kernel_d_2<Coeff> is based on an algorithm computing a geometric-topological analysis of a single curve [EKW07] and of a pair of curves [EK08]. The main idea behind both analyses is to compute the critical x-coordinates of curves and curve pairs by projection (resultants), and compute additional information about the critical fibers using subresultants and Sturm-Habicht sequences [GVRLR98]. With that information, the fiber at critical x-coordinates is computed by a variant of the Bitstream Descartes method. See also [Ker09] for a comprehensive description of these techniques. Almost all functors in the class that take a *Polynomial_2* object as argument trigger such an analysis as a main computation step. For efficiency, these analyses (of single curves and curve pairs) are therefore cached internally for efficiency. For instance, computing the pairwise solutions of 10 *Polynomial_2* objects requires 10 curve analyses and 45 curve pair analyses to be computed internally.

A point p of type *Algebraic_real_2* is represented by its x-coordinate x_0 (as described in the *Algebraic_kernel_d_1* paragraph above), an algebraic curve where p lies on, and an integer i, denoting that p is the ith point in the fiber at x_0, counted from the bottom (ignoring a possible vertical line at x_0). Note that this determines the point uniquely, but the y-coordinate is not stored internally in terms of an *Algebraic_real_1* object. Querying such a representation by calling *Compute_y_2* is a time-consuming step, and should be avoided for efficiency reasons if possible.

8.3.2 Algebraic Kernels Based on RS

The package offers two univariate algebraic kernels that are based on the library Rs [RS], namely *CGAL::Algebraic_kernel_rs_gmpz_d_1* and *CGAL::Algebraic_kernel_rs_gmpq_d_1*. As the names indicate, the kernels are based on the library Rs [RS] and support univariate polynomials over *CGAL::Gmpz* or *CGAL::Gmpq*, respectively.

In general we encourage to use *CGAL::Algebraic_kernel_rs_gmpz_d_1* instead of *CGAL::Algebraic_kernel_rs_gmpq_d_1*. This is caused by the fact that the most efficient way to compute operations (such as gcd) on polynomials with rational coefficients is to use the corresponding implementation for polynomials with integer coefficients. That is, the *CGAL::Algebraic_kernel_rs_gmpq_d_1* is slightly slower due to overhead caused by the necessary conversions. However, since this may not always be a major issue, the *CGAL::Algebraic_kernel_rs_gmpq_d_1* is provided for convenience.

The core of both kernels is the implementation of the interval Descartes algorithm [RZ04] of the library Rs [RS], which is used to isolate the roots of the polynomial. The Rs library restricts its attention to univariate integer polynomials and some substantial gain of efficiency can be made by using a kernel that does not follow the generic programming paradigm, by avoiding interfaces between layers. Specifically, working with only one number type allows to optimize some polynomial operations as well as memory handling. The implementation of these kernels make heavy use of the MPFR [MPFb] and MPFI [MPFa] libraries, and of their CGAL interfaces, *Gmpfr* and *Gmpfi*. The algebraic numbers (roots of the polynomials) are represented in the two Rs kernels by a *Gmpfi* interval and a pointer to the polynomial of which they are roots. See [LPT09] for more details on the implementation, tests of these kernels, comparisons with other algebraic kernels and discussions about the efficiency.

8.4 Examples

Construction of Algebraic Real Numbers

The following example illustrates the construction of *AlgebraicKernel_d_1::Algebraic_real_1* using *AlgebraicKernel_d_1::Construct_algebraic_real_1*:

```
#include <CGAL/basic.h>
#ifdef CGAL_USE_MPFI
#include <CGAL/Algebraic_kernel_d_1.h>
#include <CGAL/Gmpz.h>
#include <vector>
#include <iostream>

typedef CGAL::Algebraic_kernel_d_1<CGAL::Gmpz>          AK;
typedef AK::Polynomial_1                                Polynomial_1;
typedef AK::Algebraic_real_1                            Algebraic_real_1;
typedef AK::Coefficient                                 Coefficient;
typedef AK::Bound                                       Bound;
typedef AK::Multiplicity_type                           Multiplicity_type;

int main(){
  AK ak; // an object of
  AK::Construct_algebraic_real_1 construct_algreal_1 =
    ak.construct_algebraic_real_1_object();

  std::cout << "Construct from int         : " << construct_algreal_1(int(2)) <<
    "\n";
  std::cout << "Construct from Coefficient : " <<
    construct_algreal_1(Coefficient(2)) << "\n";
  std::cout << "Construct from Bound       : " << construct_algreal_1(Bound(2)) <<
     "\n\n";

  Polynomial_1 x = CGAL::shift(AK::Polynomial_1(1),1); // the monomial x
  std::cout << "Construct by index              : "
            << construct_algreal_1(x*x-2,1) << "\n"
            << to_double(construct_algreal_1(x*x-2,1)) << "\n";
  std::cout << "Construct by isolating interval : "
            << construct_algreal_1(x*x-2,Bound(0),Bound(2)) << "\n"
            << to_double(construct_algreal_1(x*x-2,Bound(0),Bound(2))) << "\n\n";

  return 0;
}
#else
int main(){
  std::cout << "This example requires CGAL to be configured with library MPFI." <<
     std::endl;
return 0;
}
#endif
```

File: examples/Algebraic_kernel_d/Construct_algebraic_real_1.cpp

Solving Univariate Polynomials

The following example illustrates the construction of *AlgebraicKernel_d_1::Algebraic_real_1* using *AlgebraicKernel_d_1::Solve_1*:

```
#include <CGAL/basic.h>
#ifdef CGAL_USE_MPFI
#include <CGAL/Algebraic_kernel_d_1.h>
#include <CGAL/Gmpz.h>
#include <vector>

typedef CGAL::Algebraic_kernel_d_1<CGAL::Gmpz>          AK;
typedef AK::Polynomial_1                                 Polynomial_1;
typedef AK::Algebraic_real_1                             Algebraic_real_1;
typedef AK::Bound                                        Bound;
typedef AK::Multiplicity_type                            Multiplicity_type;

int main(){
  AK ak; // an object of
  AK::Solve_1 solve_1 = ak.solve_1_object();
  Polynomial_1 x = CGAL::shift(AK::Polynomial_1(1),1); // the monomial x


  // variant using a bool indicating a square free polynomial
  // multiplicities are not computed
  std::vector<Algebraic_real_1> roots;
  solve_1(x*x-2,true, std::back_inserter(roots));
  std::cout << "Number of roots is           : " << roots.size() << "\n";
  std::cout << "First root should be -sqrt(2): " << CGAL::to_double(roots[0]) <<
    "\n";
  std::cout << "Second root should be sqrt(2): " << CGAL::to_double(roots[1]) <<
    "\n\n";
  roots.clear();

  // variant for roots in a given range of a square free polynomial
  solve_1((x*x-2)*(x*x-3),true, Bound(0),Bound(10),std::back_inserter(roots));
  std::cout << "Number of roots is           : " << roots.size() << "\n";
  std::cout << "First root should be  sqrt(2): " << CGAL::to_double(roots[0]) <<
    "\n";
  std::cout << "Second root should be sqrt(3): " << CGAL::to_double(roots[1]) <<
    "\n\n";
  roots.clear();

  // variant computing all roots with multiplicities
  std::vector<std::pair<Algebraic_real_1,Multiplicity_type> > mroots;
  solve_1((x*x-2), std::back_inserter(mroots));
  std::cout << "Number of roots is           : " << mroots.size() << "\n";
  std::cout << "First root should be -sqrt(2): " <<
    CGAL::to_double(mroots[0].first) << ""
            << " with multiplicity "           << mroots[0].second << "\n";
  std::cout << "Second root should be sqrt(2): " <<
    CGAL::to_double(mroots[1].first) << ""
            << " with multiplicity "           << mroots[1].second << "\n\n";
  mroots.clear();

  // variant computing roots with multiplicities for a range
  solve_1((x*x-2)*(x*x-3),Bound(0),Bound(10),std::back_inserter(mroots));
  std::cout << "Number of roots is           : " << mroots.size() << "\n";
```

```
  std::cout << "First root should be  sqrt(2): " <<
    CGAL::to_double(mroots[0].first) << ""
            << " with multiplicity "              << mroots[0].second << "\n";
  std::cout << "Second root should be sqrt(3): " <<
    CGAL::to_double(mroots[1].first) << ""
            << " with multiplicity "              << mroots[1].second << "\n\n";
  return 0;
}
#else
int main(){
  std::cout << "This example requires CGAL to be configured with library MPFI." <<
     std::endl;
return 0;
}
#endif
```

File: examples/Algebraic_kernel_d/Solve_1.cpp

Comparison and Approximation of Algebraic Real Numbers

The following example illustrates the comparison of *AlgebraicKernel_d_1::Algebraic_real_1* numbers:

```
#include <CGAL/basic.h>
#ifdef CGAL_USE_MPFI
#include <CGAL/Algebraic_kernel_d_1.h>
#include <CGAL/Gmpz.h>
#include <vector>

typedef CGAL::Algebraic_kernel_d_1<CGAL::Gmpz>          AK;
typedef AK::Coefficient                                 Coefficient;
typedef AK::Polynomial_1                                Polynomial_1;
typedef AK::Algebraic_real_1                            Algebraic_real_1;
typedef AK::Bound                                       Bound;
typedef std::pair<Bound,Bound>                          Interval;

int main(){
  AK ak;

  AK::Construct_algebraic_real_1 construct_algebraic_real_1 =
    ak.construct_algebraic_real_1_object();
  Polynomial_1 x = CGAL::shift(AK::Polynomial_1(1),1); // the monomial x
  Algebraic_real_1 a = construct_algebraic_real_1(x*x-2,1); //  sqrt(2)
  Algebraic_real_1 b = construct_algebraic_real_1(x*x-3,1); //  sqrt(3)

  // Algebraic_real_1 is RealEmbeddable (just some functions:)
  std::cout << "sign of a is                 : " << CGAL::sign(a)      << "\n";
  std::cout << "double approximation of a is : " << CGAL::to_double(a) << "\n";
  std::cout << "double approximation of b is : " << CGAL::to_double(b) << "\n";
  std::cout << "double lower bound of a      : " << CGAL::to_interval(a).first  <<
     "\n";
  std::cout << "double upper bound of a      : " << CGAL::to_interval(a).second <<
     "\n";
  std::cout << "LessThanComparable (a<b)     : " << (a<b) << "\n\n";

  // use compare_1 with int, Bound, Coefficient, Algebraic_real_1
  AK::Compare_1 compare_1 = ak.compare_1_object();
  std::cout << " compare with an int                  : " << compare_1(a ,int(2))
    << "\n";
  std::cout << " compare with an Coefficient          : " << compare_1(a
    ,Coefficient(2)) << "\n";
  std::cout << " compare with an Bound                : " << compare_1(a
    ,Bound(2)) << "\n";
  std::cout << " compare with another Algebraic_real_1: " << compare_1(a ,b) <<
    "\n\n";

  // get a value between two roots
  AK::Bound_between_1 bound_between_1 = ak.bound_between_1_object();
  std::cout << " value between sqrt(2) and sqrt(3) " << bound_between_1(a,b) <<
    "\n";
  std::cout << " is larger than sqrt(2)             " <<
    compare_1(bound_between_1(a,b),a) << "\n";
  std::cout << " is less   than sqrt(3)             " <<
    compare_1(bound_between_1(a,b),b) << "\n\n";

  // approximate with relative precision
  AK::Approximate_relative_1 approx_r = ak.approximate_relative_1_object();
```

```
  std::cout << " lower bound of a with at least 100 bits:    "<<
    approx_r(a,100).first  << "\n";
  std::cout << " upper bound of a with at least 100 bits:    "<<
    approx_r(a,100).second << "\n\n";

  // approximate with absolute error
  AK::Approximate_absolute_1 approx_a = ak.approximate_absolute_1_object();
  std::cout << " lower bound of b with error less than 2^-100:   "<<
    approx_a(b,100).first  << "\n";
  std::cout << " upper bound of b with error less than 2^-100:   "<<
    approx_a(b,100).second << "\n\n";

  return 0;
  }
#else
int main(){
  std::cout << "This example requires CGAL to be configured with library MPFI." <<
     std::endl;
return 0;
}
#endif
```

File: examples/Algebraic_kernel_d/Compare_1.cpp

Isolation of Algebraic Real Numbers with respect to roots of other polynomials

The following example illustrates the isolation of *AlgebraicKernel_d_1::Algebraic_real_1* numbers:

```
#include <CGAL/basic.h>
#ifdef CGAL_USE_MPFI
#include <CGAL/Algebraic_kernel_d_1.h>
#include <CGAL/Gmpz.h>
#include <vector>

typedef CGAL::Algebraic_kernel_d_1<CGAL::Gmpz>          AK;
typedef AK::Polynomial_1                                Polynomial_1;
typedef AK::Algebraic_real_1                            Algebraic_real_1;
typedef AK::Coefficient                                 Coefficient;
typedef AK::Bound                                       Bound;
typedef AK::Multiplicity_type                           Multiplicity_type;

int main(){
  AK ak; // an object of
  AK::Construct_algebraic_real_1 construct_algreal_1 =
    ak.construct_algebraic_real_1_object();
  AK::Isolate_1 isolate_1 = ak.isolate_1_object();
  AK::Compute_polynomial_1 compute_polynomial_1 =
    ak.compute_polynomial_1_object();

  // construct an algebraic number from an integer
  Algebraic_real_1 frominteger=construct_algreal_1(int(2));
  std::cout << "Construct from int: " << frominteger << "\n";

  // the constructed algebraic number is root of a polynomial
  Polynomial_1 pol=compute_polynomial_1(frominteger);
  std::cout << "The constructed number is root of: " << pol << "\n";

  // construct an algebraic number from a polynomial and an isolating interval
  Polynomial_1 x = CGAL::shift(AK::Polynomial_1(1),1); // the monomial x
  Algebraic_real_1 frominterval=construct_algreal_1(x*x-2,Bound(0),Bound(2));
  std::cout << "Construct from isolating interval: " << frominterval << "\n";

  // isolate the second algebraic number from the first: this is to say,
  // isolating the second algebraic number with respect to the polynomial
  // of which the first constructed number is root
  std::pair<Bound,Bound> isolation1 = isolate_1(frominterval,pol);
  std::cout << "Isolating the second algebraic number gives: ["
            << isolation1.first << "," << isolation1.second << "]\n";

  // isolate again the same algebraic number, this time with respect to
  // the polynomial 10*x-14 (which has root 1.4, close to this algebraic
  // number)
  std::pair<Bound,Bound> isolation2 = isolate_1(frominterval,10*x-14);
  std::cout << "Isolating again the second algebraic number gives: ["
            << isolation2.first << "," << isolation2.second << "]\n";

  return 0;
}
#else
int main(){
  std::cout << "This example requires CGAL to be configured with library MPFI." <<
     std::endl;
```

```
return 0;
}
#endif
```

File: examples/Algebraic_kernel_d/Isolate_1.cpp

Interplay with Polynomials

The following example illustrates the sign evaluation of *AlgebraicKernel_d_1::Algebraic_real_1* numbers in polynomials:

```
#include <CGAL/basic.h>
#ifdef CGAL_USE_MPFI
#include <CGAL/Algebraic_kernel_d_1.h>
#include <CGAL/Gmpz.h>
#include <vector>

typedef CGAL::Algebraic_kernel_d_1<CGAL::Gmpz>          AK;
typedef AK::Polynomial_1                                Polynomial_1;
typedef AK::Algebraic_real_1                            Algebraic_real_1;
typedef AK::Coefficient                                 Coefficient;
typedef AK::Bound                                       Bound;
typedef AK::Multiplicity_type                           Multiplicity_type;

int main(){
  AK ak;
  AK::Construct_algebraic_real_1 construct_algreal_1 =
    ak.construct_algebraic_real_1_object();
  AK::Solve_1 solve_1 = ak.solve_1_object();
  AK::Sign_at_1 sign_at_1 = ak.sign_at_1_object();
  AK::Is_zero_at_1 is_zero_at_1 = ak.is_zero_at_1_object();

  // construct the polynomials p=x^2-5 and q=x-2
  Polynomial_1 x = CGAL::shift(AK::Polynomial_1(1),1); // the monomial x
  Polynomial_1 p = x*x-5;
  std::cout << "Polynomial p: " << p << "\n";
  Polynomial_1 q = x-2;
  std::cout << "Polynomial q: " << q << "\n";

  // find the roots of p (it has two roots) and q (one root)
  std::vector<Algebraic_real_1> roots_p,roots_q;
  solve_1(p,true, std::back_inserter(roots_p));
  solve_1(q,true, std::back_inserter(roots_q));

  // evaluate the second root of p in q
  std::cout << "Sign of the evaluation of root 2 of p in q: "
            << sign_at_1(q,roots_p[1]) << "\n";

  // evaluate the root of q in p
  std::cout << "Sign of the evaluation of root 1 of q in p: "
            << sign_at_1(p,roots_q[0]) << "\n";

  // check whether the evaluation of the first root of p in p is zero
  std::cout << "Is zero the evaluation of root 1 of p in p? "
            << is_zero_at_1(p,roots_p[0]) << "\n";

  return 0;
}

#else
int main(){
  std::cout << "This example requires CGAL to be configured with library MPFI." <<
     std::endl;
return 0;
```

```
}
#endif
```

File: examples/Algebraic_kernel_d/Sign_at_1.cpp

8.5 Design and Implementation History

This package is clearly split into a univariate and bivariate kernel. However, with respect to its history the package splits into a design part and an implementation part.

The concepts, which make up the design part, were written by Eric Berberich, Michael Hemmer, and Monique Teillaud. The design history of the package is fairly old and several ideas that influenced this package can already be found in [BHKT07]. Since then, the initial design underwent considerable changes. For instance, it was decided that the algebraic numbers should be under the control of the algebraic kernel. On the other hand the initial support for polynomials was extended to a separate and independent package that is not restricted to a certain number of variables. Thus, the authors want to thank for all the useful feedback and ideas that was brought to them throughout the last years. In particular, they want to thank Menelaos Karavelas and Elias Tsigaridas for their initial contributions.

The two generic models where initially developed as part of the EXACUS [BEH+05] project. However, the models are now fully integrated into the CGAL library, since also the relevant layers of EXACUS are now part of CGAL. The main authors for *CGAL::Algebraic_kernel_d_1<Coeff>* and *CGAL::Algebraic_kernel_d_2<Coeff>* are Michael Hemmer and Michael Kerber, respectively. Notwithstanding, the authors also want to emphasize the contribution of all authors of the EXACUS project, particularly the contribution of Arno Eigenwillig, Sebastian Limbach and Pavel Emeliyanenko.

The two univariate kernels that interface the library Rs [RS] were written by Luis Peñaranda and Sylvain Lazard. Both models interface the library Rs [RS] by Fabrice Rouillier. The authors want to thank Fabrice Rouillier and Elias Tsigaridas for strong support and many useful discussions that lead to the integration of Rs.

Algebraic Kernel
Reference Manual

8.6 Classified Reference Pages

8.6.1 Concepts

Univariate Algebraic Kernel

Bivariate Algebraic Kernel

8.6.2 Models

8.7 Alphabetical List of Reference Pages

Concept

AlgebraicKernel_d_1

Definition

A model of the *AlgebraicKernel_d_1* concept is meant to provide the algebraic functionalities on univariate polynomials of general degree d.

Refines

CopyConstructible
Assignable

A model of *AlgebraicKernel_d_1* must provide:

Types

AlgebraicKernel_d_1:: Coefficient	A model of *IntegralDomain* and *RealEmbeddable*. *ExplicitInteroperable* with *AlgebraicKernel_d_1::Bound*.
AlgebraicKernel_d_1:: Polynomial_1	A univariate polynomial that is a model of *Polynomial_d*, where *CGAL::Polynomial_traits_d<Polynomial_1> ::Innermost_coefficient* is *AlgebraicKernel_d_1::Coefficient*.
AlgebraicKernel_d_1:: Algebraic_real_1	A type that is used to represent real roots of univariate polynomials. The type must be a model of *DefaultConstructible*, *CopyConstructible*, *Assignable* and *RealEmbeddable*.
AlgebraicKernel_d_1:: Bound	A type to represent upper and lower bounds of *AlgebraicKernel_d_1::Algebraic_real_1*. The type is *ExplicitInteroperable* with *AlgebraicKernel_d_1::Coefficient* and must be a model *IntegralDomain*, *RealEmbeddable* and dense in $\mathbb{R}$.
AlgebraicKernel_d_1:: size_type	Size type (unsigned integral type).
AlgebraicKernel_d_1:: Multiplicity_type	Multiplicity type (unsigned integral type).

Functors

AlgebraicKernel_d_1:: Construct_algebraic_real_1

A model of *AlgebraicKernel_d_1::ConstructAlgebraicReal_1*.

AlgebraicKernel_d_1:: Compute_polynomial_1	A model of *AlgebraicKernel_d_1::ComputePolynomial_1*.
AlgebraicKernel_d_1:: Isolate_1	A model of *AlgebraicKernel_d_1::Isolate_1*.

AlgebraicKernel_d_1:: Is_square_free_1	A model of *AlgebraicKernel_d_1::IsSquareFree_1*.
AlgebraicKernel_d_1:: Make_square_free_1	A model of *AlgebraicKernel_d_1::MakeSquareFree_1*.

AlgebraicKernel_d_1:: Square_free_factorize_1

A model of *AlgebraicKernel_d_1::SquareFreeFactorize_1*.

AlgebraicKernel_d_1:: Is_coprime_1	A model of *AlgebraicKernel_d_1::IsCoprime_1*.
AlgebraicKernel_d_1:: Make_coprime_1	A model of *AlgebraicKernel_d_1::MakeCoprime_1*.
AlgebraicKernel_d_1:: Solve_1	A model of *AlgebraicKernel_d_1::Solve_1*.
AlgebraicKernel_d_1:: Number_of_solutions_1	A model of *AlgebraicKernel_d_1::NumberOfSolutions_1*.
AlgebraicKernel_d_1:: Sign_at_1	A model of *AlgebraicKernel_d_1::SignAt_1*.
AlgebraicKernel_d_1:: Compare_1	A model of *AlgebraicKernel_d_1::Compare_1*.
AlgebraicKernel_d_1:: Bound_between_1	A model of *AlgebraicKernel_d_1::BoundBetween_1*.

AlgebraicKernel_d_1:: Approximate_absolute_1

A model of *AlgebraicKernel_d_1::ApproximateAbsolute_1*.

AlgebraicKernel_d_1:: Approximate_relative_1

A model of *AlgebraicKernel_d_1::ApproximateRelative_1*.

Operations

For each of the function objects above, there must exist a member function that requires no arguments and returns an instance of that function object. The name of the member function is the uncapitalized name of the type returned with the suffix *_object* appended. For example, for the function object *AlgebraicKernel_d_1::Bound_between_1* the following member function must exist:

AlgebraicKernel_d_1::Bound_between_1

ak_1.bound_between_1_object() const

Has Models

Algebraic_kernel_rs_gmpz_d_1
Algebraic_kernel_rs_gmpq_d_1

See Also

AlgebraicKernel_d_2 . . . . page 422

Functor Concept

AlgebraicKernel_d_1::ConstructAlgebraicReal_1

Definition

Constructs *AlgebraicKernel_d_1::Algebraic_real_1*.

Refines

AdaptableFunctor

Types

typedef AlgebraicKernel_d_1::Algebraic_real_1 *result_type;*

Operations

result_type *fo(int a)* introduces an *AlgebraicKernel_d_1::Algebraic_real_1* initialized to a.

result_type *fo(AlgebraicKernel_d_1::Bound a)*

introduces an *AlgebraicKernel_d_1::Algebraic_real_1* initialized to a.

result_type *fo(AlgebraicKernel_d_1::Coefficient a)*

introduces an *AlgebraicKernel_d_1::Algebraic_real_1* initialized to a.

result_type *fo(AlgebraicKernel_d_1::Polynomial_1 p, AlgebraicKernel_d_1::size_type i)*

introduces an *AlgebraicKernel_d_1::Algebraic_real_1* initialized to the i-th real root of p. The index starts at 0, that is, p must have at least $i+1$ real roots.
Precondition: p is square free.
Precondition: p has at least $i+1$ real roots.

result_type *fo.operator()(AlgebraicKernel_d_1::Polynomial_1 p,
AlgebraicKernel_d_1::Bound l,
AlgebraicKernel_d_1::Bound u)*

introduces an *AlgebraicKernel_d_1::Algebraic_real_1* initialized to the only real root of p in the open interval $I = (l, u)$.
Precondition: $l < u$
Precondition: p is square free.
Precondition: p has exactly one real root in I
Precondition: p has no real root on ∂I

See Also

AlgebraicKernel_d_2::ConstructAlgebraicReal_2 . . . . page 424

Functor
Concept

AlgebraicKernel_d_1::ComputePolynomial_1

Definition

Computes a square free univariate polynomial p, such that the given *AlgebraicKernel_d_1::Algebraic_real_1* is a root of p.

Refines

AdaptableUnaryFunction

Types

typedef AlgebraicKernel_d_1::Polynomial_1	*result_type;*
typedef AlgebraicKernel_d_1::Algebraic_real_1	*argument_type;*

Operations

result_type *fo(argument_type x)*

Computes a square free polynomial p, such that x is a real root of p.

See Also

AlgebraicKernel_d_1::Isolate_1 . . . . page 403

AlgebraicKernel_d_1::Isolate_1

Functor
Concept

Definition

Computes an open isolating interval for an *AlgebraicKernel_d_1::Algebraic_real_1* with respect to the real roots of a given univariate polynomial.

Refines

AdaptableBinaryFunction

Types

typedef std::pair<AlgebraicKernel_d_1::Bound,AlgebraicKernel_d_1::Bound> *result_type;*

typedef AlgebraicKernel_d_1::Algebraic_real_1 *first_argument_type;*

typedef AlgebraicKernel_d_1::Polynomial_1 *second_argument_type;*

Operations

result_type *fo(first_argument_type a, second_argument_type p)*

Computes an open isolating interval $I = (l, u)$ for a with respect to the real roots of p. It is not required that a is a root of p.
Postcondition: $a \in I$.
Postcondition: $p(x) \neq 0 | \forall x \in \bar{I} \backslash a$.

See Also

AlgebraicKernel_d_1::ComputePolynomial_1 . . . . page 402

Functor
Concept

AlgebraicKernel_d_1::IsSquareFree_1

Definition

Computes whether the given univariate polynomial is square free.

Refines

AdaptableUnaryFunction

Types

typedef bool	*result_type;*
typedef AlgebraicKernel_d_1::Polynomial_1	*argument_type;*

A model of this type must provide:

Operations

result_type *fo(argument_type p)*

Returns true if p is square free.

See Also

AlgebraicKernel_d_1::MakeSquareFree_1 .... page 405
AlgebraicKernel_d_1::SquareFreeFactorize_1 .... page 406

AlgebraicKernel_d_1::MakeSquareFree_1

Functor
Concept

Definition

Returns a square free part of a univariate polynomial.

Refines

AdaptableUnaryFunction

Types

typedef AlgebraicKernel_d_1::Polynomial_1	*result_type;*
typedef AlgebraicKernel_d_1::Polynomial_1	*argument_type;*

Operations

result_type *fo(argument_type p)*

Returns a square free part of p

See Also

AlgebraicKernel_d_1::IsSquareFree_1 . . . . page 404
AlgebraicKernel_d_1::SquareFreeFactorize_1 . . . . page 406

Functor
Concept

AlgebraicKernel_d_1::SquareFreeFactorize_1

Definition

Computes a square free factorization of an *AlgebraicKernel_d_1::Polynomial_1*.

A polynomial p is factored into square free and pairwise coprime non-constant factors q_i with multiplicities m_i and a constant factor c, such that $p = c \cdot q_1^{m_1} \cdot \ldots \cdot q_n^{m_n}$.

The factor multiplicity pairs $< q_i, m_i >$ are written to the given output iterator. The constant factor c is not computed.

Refines

Assignable
CopyConstructible

Operations

template < class OutputIterator >
OutputIterator *fo(AlgebraicKernel_d_1::Polynomial_1 p, OutputIterator res)*

Copies in the output iterator the factors of a square free factorization of p, with their multiplicity, as objects of type *std::pair<AlgebraicKernel_d_1::Polynomial_1, AlgebraicKernel_d_1::Multiplicity_type>*.

See Also

AlgebraicKernel_d_1::IsSquareFree_1 . . . . page 404
AlgebraicKernel_d_1::MakeSquareFree_1 . . . . page 405

AlgebraicKernel_d_1::IsCoprime_1

Functor
Concept

Definition

Determines whether a given pair of univariate polynomials p_1, p_2 is coprime, namely if $\deg(\gcd(p_1, p_2)) = 0$.

Refines

AdaptableBinaryFunction

Types

typedef bool	*result_type;*
typedef AlgebraicKernel_d_1::Polynomial_1	*first_argument_type;*
typedef AlgebraicKernel_d_1::Polynomial_1	*second_argument_type;*

Operations

result_type *fo(first_argument_type p1, second_argument_type p2)*

Returns true if *p1* and *p2* are coprime.

See Also

AlgebraicKernel_d_1::MakeCoprime_1 . . . . page 408

Functor Concept

AlgebraicKernel_d_1::MakeCoprime_1

Definition

Computes for a given pair of univariate polynomials p_1, p_2 their common part g up to a constant factor and coprime parts q_1, q_2 respectively.

That is, it computes g, q_1, q_2 such that:
$c_1 \cdot p_1 = g \cdot q_1$ for some constant c_1 and
$c_2 \cdot p_2 = g \cdot q_2$ for some constant c_2, such that q_1 and q_2 are coprime.

It returns true if p_1 and p_2 are already coprime.

Refines

AdaptableFunctor with five arguments

Types

typedef bool *result_type;*

Operations

result_type *fo.operator()(AlgebraicKernel_d_1::Polynomial_1 p1,*
AlgebraicKernel_d_1::Polynomial_1 p2,
AlgebraicKernel_d_1::Polynomial_1 & g,
AlgebraicKernel_d_1::Polynomial_1 & q1,
AlgebraicKernel_d_1::Polynomial_1 & q2)

Computes g, q_1, q_2 as described above.
Returns whether p_1 and p_2 where already coprime.

See Also

AlgebraicKernel_d_1::IsCoprime_1 . . . . page 407

AlgebraicKernel_d_1::Solve_1

Definition

Computes the real roots of a univariate polynomial.

Refines

Assignable
CopyConstructible

Operations

A model of this type must provide:

template < class OutputIterator >
OutputIterator *fo(AlgebraicKernel_d_1::Polynomial_1 p, OutputIterator res)*

Computes all real solutions of p with multiplicity, and copies them as objects of type *std::pair<AlgebraicKernel_d_1::Algebraic_real_1, AlgebraicKernel_d_1::Multiplicity_type>* in *res*.

template < class OutputIterator >
OutputIterator *fo.operator()(AlgebraicKernel_d_1::Polynomial_1 p,*
bool known_to_be_square_free,
OutputIterator res)

Computes all real solutions of p, and copies them as objects of type *AlgebraicKernel_d_1::Algebraic_real_1* in *res*. The bool *known_to_be_square_free* indicates whether p is known to be square free. Each root, though it might be a multiple root, is reported only once.

template < class OutputIterator >
OutputIterator *fo.operator()(AlgebraicKernel_d_1::Polynomial_1 p,*
AlgebraicKernel_d_1::Bound l,
AlgebraicKernel_d_1::Bound u,
OutputIterator res)

Computes all real solutions of p in the closed interval $[l,u]$ with multiplicity, and copies them as objects of type *std::pair<AlgebraicKernel_d_1::Algebraic_real_1, AlgebraicKernel_d_1::Multiplicity_type>* in *res*.

template < class OutputIterator >
OutputIterator *fo.operator()(AlgebraicKernel_d_1::Polynomial_1 p,*
bool known_to_be_square_free,
AlgebraicKernel_d_1::Bound l,
AlgebraicKernel_d_1::Bound u,

OutputIterator res)

Computes all real solutions of p in the closed interval $[l,u]$, and copies them as objects of type *AlgebraicKernel_d_1::Algebraic_real_1* in *res*. The *bool known_to_be_square_free* indicates whether p is known to be square free. Each root, though it might be a multiple root, is reported only once.

See Also

AlgebraicKernel_d_1::NumberOfSolutions_1

Functor
Concept

Definition

Computes the number of real solutions of the given univariate polynomial.

Refines

AdaptableUnaryFunction

Types

typedef AlgebraicKernel_d_1::size_type	*result_type;*
typedef AlgebraicKernel_d_1::Polynomial_1	*argument_type;*

A model of this type must provide:

Operations

result_type   *fo(argument_type p)*

Returns the number of real solutions of p.
Precondition: p is square free.

See Also

AlgebraicKernel_d_1::ConstructAlgebraicReal_1 . . . . page 400

Functor Concept

AlgebraicKernel_d_1::SignAt_1

Definition

Computes the sign of a univariate polynomial *AlgebraicKernel_d_1::Polynomial_1* at a real value of type *AlgebraicKernel_d_1::Algebraic_real_1*.

Refines

AdaptableBinaryFunction

Types

AlgebraicKernel_d_1::SignAt_1:: result_type	Type convertible to *CGAL::Sign*
typedef AlgebraicKernel_d_1::Polynomial_1	*first_argument_type;*
typedef AlgebraicKernel_d_1::Algebraic_real_1	*second_argument_type;*

Operations

result_type *fo(first_argument_type p, second_argument_type x)*

Computes the sign of p at x.

See Also

AlgebraicKernel_d_1::IsZeroAt_1 . . . . page 413

AlgebraicKernel_d_1::IsZeroAt_1

Functor Concept

Definition

Computes whether an *AlgebraicKernel_d_1::Polynomial_1* is zero at a given *AlgebraicKernel_d_1::Algebraic_real_1*.

Refines

AdaptableBinaryFunction

Types

AlgebraicKernel_d_1::IsZeroAt_1:: result_type

Type convertible to *bool*

typedef AlgebraicKernel_d_1::Polynomial_1	*first_argument_type;*
typedef AlgebraicKernel_d_1::Algebraic_real_1	*second_argument_type;*

Operations

result_type *fo(first_argument_type p, second_argument_type x)*

Computes whether p is zero at x.

See Also

AlgebraicKernel_d_1::SignAt_1 . . . . page 412

Functor
Concept

AlgebraicKernel_d_1::Compare_1

Definition

Compares *AlgebraicKernel_d_1::Algebraic_real_1* values.

Refines

AdaptableBinaryFunction

Types

AlgebraicKernel_d_1::Compare_1:: result_type

Type convertible to *CGAL::Comparison_result*

typedef AlgebraicKernel_d_1::Algebraic_real_1 *first_argument_type;*
typedef AlgebraicKernel_d_1::Algebraic_real_1 *second_argument_type;*

Operations

The following operators and their symmetric counterparts are required:

result_type *fo(AlgebraicKernel_d_1::Algebraic_real_1 a, AlgebraicKernel_d_1::Algebraic_real_1 b)*

Compares *a* and *b*.

result_type *fo(AlgebraicKernel_d_1::Algebraic_real_1 a, int b)*

Compares *a* and *b*.

result_type *fo(AlgebraicKernel_d_1::Algebraic_real_1 a, AlgebraicKernel_d_1::Bound b)*

Compares *a* and *b*.

result_type *fo(AlgebraicKernel_d_1::Algebraic_real_1 a, AlgebraicKernel_d_1::Coefficient b)*

Compares *a* and *b*.

AlgebraicKernel_d_1::BoundBetween_1

Functor
Concept

Definition

Computes a number of type *AlgebraicKernel_d_1::Bound* in-between two *AlgebraicKernel_d_1::Algebraic_real_1* values.

Refines

AdaptableBinaryFunction

Types

typedef AlgebraicKernel_d_1::Bound	*result_type;*
typedef AlgebraicKernel_d_1::Algebraic_real_1	*first_argument_type;*
typedef AlgebraicKernel_d_1::Algebraic_real_1	*second_argument_type;*

Operations

result_type *fo(first_argument_type a, second_argument_type b)*

Computes a value r, which is between a and b.

Precondition: $a \neq b$
Postcondition: $r > min(a,b)$
Postcondition: $r < max(a,b)$

Functor
Concept

AlgebraicKernel_d_1::ApproximateAbsolute_1

Definition

A model of AlgebraicKernel_d_1::ApproximateAbsolute_1 is an *AdaptableBinaryFunction* that computes an approximation of an *AlgebraicKernel_d_1::Algebraic_real_1* value with respect to a given absolute precision.

Refines

AdaptableBinaryFunction

Types

typedef std::pair<AlgebraicKernel_d_1::Bound, AlgebraicKernel_d_1::Bound>

result_type;

typedef AlgebraicKernel_d_1::Algebraic_real_1 *first_argument_type;*
typedef int *second_argument_type;*

Operations

result_type *fo(first_argument_type x, second_argument_type a)*

The function computes a pair p of *AlgebraicKernel_d_1::Bound*, where $p.first$ represents the lower approximation and $p.second$ represents the upper approximation. The pair p approximates the given value x with respect to the given absolute precision a.
Postcondition: $p.first <= x$
Postcondition: $x <= p.second$
Postcondition: $(x - p.first) <= 2^{-a}$
Postcondition: $(p.second - x) <= 2^{-a}$

See Also

AlgebraicKernel_d_1::ApproximateRelative_1 . . . . page 417

AlgebraicKernel_d_1::ApproximateRelative_1

Functor Concept

Definition

A model of AlgebraicKernel_d_1::ApproximateRelative_1 is an *AdaptableBinaryFunction* that computes an approximation of an *AlgebraicKernel_d_1::Algebraic_real_1* value with respect to a given relative precision.

Refines

AdaptableBinaryFunction

Types

typedef std::pair<AlgebraicKernel_d_1::Bound, AlgebraicKernel_d_1::Bound>
result_type;

typedef AlgebraicKernel_d_1::Algebraic_real_1	*first_argument_type;*
typedef int	*second_argument_type;*

Operations

result_type *fo(first_argument_type x, second_argument_type r)*

The function computes a pair p of *AlgebraicKernel_d_1::Bound*, where $p.first$ represents the lower approximation and $p.second$ represents the upper approximation. The pair p approximates the given value x with respect to the given relative precision r.
Postcondition: $p.first <= x$
Postcondition: $x <= p.second$
Postcondition: $(x - p.first) <= 2^{-r} * |x|$
Postcondition: $(p.second - x) <= 2^{-r} * |x|$

See Also

AlgebraicKernel_d_1::ApproximateAbsolute_1 .... page 416

Class

CGAL::Algebraic_kernel_d_1<Coeff>

#include <CGAL/Algebraic_kernel_d_1.h>

Definition

The class represents an algebraic real root by a square free polynomial and an isolating interval that uniquely defines the root. The template argument *Coeff* determines the coefficient type of the kernel, which is also the coefficient type of the supported polynomials.

Currently, the following coefficient types are supported:
– *Gmpz*, *Gmpq*, (requires configuration with external libraries GMP, MPFR and MPFI)
– *CORE::BigInt*, *CORE::BigRat*, (requires configuration with external library GMP)
– *leda_integer*, *leda_rational*. (requires configuration with external library LEDA)

advanced

The template argument type can also be set to *Sqrt_extension<NT,ROOT>*, where *NT* is one of the types listed above. *ROOT* should be one of the integer types. See also the documentation of *Sqrt_extension<NT,ROOT>*.

advanced

The current method to isolate roots is the bitstream Descartes method presented in [Eig08]. The used method to refine the approximation of an algebraic real root is a slightly modified (filtered) version of the one presented in [Abb]. The method has quadratic convergence.

Is Model for the Concepts

AlgebraicKernel_d_1.

Types

Algebraic_kernel_d_1<Coeff>:: Coefficient — Same type as the template argument *Coeff*.

Algebraic_kernel_d_1<Coeff>:: Polynomial_1 — A model of *AlgebraicKernel_d_1::Polynomial_1*.

Algebraic_kernel_d_1<Coeff>:: Algebraic_real_1

A model of *AlgebraicKernel_d_1::AlgebraicReal_1*.

Algebraic_kernel_d_1<Coeff>:: Bound — The choice of *Coeff* also determines the provided bound, type. In case of *Coeff* is:
– *Gmpz* or *Gmpq* this is *Gmpq*,
– *CORE::BigInt* or *CORE::BigInt* this is *CORE::BigRat*,
– *leda_integer* or *leda_integer* this is *leda_rational*.

Algebraic_kernel_d_1<Coeff>:: Multiplicity_type

The multiplicity type is *int*.

See Also

AlgebraicKernel_d_1 . . . . page 398
Polynomial_d . . . . page 261
CGAL::Algebraic_kernel_d_2<Coeff> . . . . page 451

Class

CGAL::Algebraic_kernel_rs_gmpz_d_1

#include <CGAL/Algebraic_kernel_rs_gmpz_d_1.h>

Definition

This univariate algebraic kernel uses the Rs library to perform integer univariate polynomial root isolation. It is a model of the *AlgebraicKernel_d_1* concept.

Types

Algebraic_kernel_rs_gmpz_d_1:: Coefficient — It is a typedef to *CGAL::Gmpz*.

Algebraic_kernel_rs_gmpz_d_1:: Polynomial_1

It is defined as *CGAL::Polynomial<CGAL::Gmpz>*.

Algebraic_kernel_rs_gmpz_d_1:: Algebraic_real_1

Type that represents the real roots of integer univariate polynomials, containing a pointer to the polynomial of which the represented algebraic number is root and and a *CGAL::Gmpfi* isolating interval.

Algebraic_kernel_rs_gmpz_d_1:: Bound — Since the isolating intervals of the roots have type *CGAL::Gmpfi*, the bounds have type *CGAL::Gmpfr*.

Algebraic_kernel_rs_gmpz_d_1:: Multiplicity_type

The multiplicity is an *int*.

Is Model for the Concepts

AlgebraicKernel_d_1.

See Also

Algebraic_kernel_rs_gmpz_d_1

CGAL::Algebraic_kernel_rs_gmpq_d_1

#include <CGAL/Algebraic_kernel_rs_gmpq_d_1.h>

Class

Definition

This univariate algebraic kernel uses the Rs library to perform rational univariate polynomial root isolation. It is a model of the *AlgebraicKernel_d_1* concept. Due to the fact that RS can only isolate integer polynomials, the operations of this kernel have the overhead of converting the polynomials to integer.

Types

Algebraic_kernel_rs_gmpq_d_1:: Coefficient It is a typedef to *CGAL::Gmpq*.

Algebraic_kernel_rs_gmpq_d_1:: Polynomial_1

It is defined as *CGAL::Polynomial<CGAL::Gmpq>*.

Algebraic_kernel_rs_gmpq_d_1:: Algebraic_real_1

Type that represents the real roots of integer univariate polynomials, containing a pointer to the polynomial of which the represented algebraic number is root and and a *CGAL::Gmpfi* isolating interval.

Algebraic_kernel_rs_gmpq_d_1:: Bound Since the isolating intervals of the roots have type *CGAL::Gmpfi*, the bounds have type *CGAL::Gmpfr*.

Algebraic_kernel_rs_gmpq_d_1:: Multiplicity_type

The multiplicity is an *int*.

Is Model for the Concepts

AlgebraicKernel_d_1

See Also

Algebraic_kernel_rs_gmpz_d_1

Concept

AlgebraicKernel_d_2

Definition

A model of the *AlgebraicKernel_d_2* concept gathers necessary tools for solving and handling bivariate polynomial systems of general degree d.

Refines

AlgebraicKernel_d_1
CopyConstructible
Assignable

Types

AlgebraicKernel_d_2:: Polynomial_2	A bivariate polynomial that is a model of *Polynomial_d*, where *CGAL::Polynomial_traits_d<Polynomial_2> ::Innermost_coefficient* is *AlgebraicKernel_d_1::Coefficient*.
AlgebraicKernel_d_2:: Algebraic_real_2	A type that is used to represent real solutions of bivariate zero dimensional polynomial systems. A model of *DefaultConstructible*, *CopyConstructible* and *Assignable*.

Functors

AlgebraicKernel_d_2:: Construct_algebraic_real_2

A model of *AlgebraicKernel_d_2::ConstructAlgebraicReal_2*.

AlgebraicKernel_d_2:: Compute_polynomial_x_2

A model of *AlgebraicKernel_d_2::ComputePolynomialX_2*.

AlgebraicKernel_d_2:: Compute_polynomial_y_2

A model of *AlgebraicKernel_d_2::ComputePolynomialY_2*.

AlgebraicKernel_d_2:: Isolate_2	A model of *AlgebraicKernel_d_2::Isolate_2*.
AlgebraicKernel_d_2:: Isolate_x_2	A model of *AlgebraicKernel_d_2::IsolateX_2*.
AlgebraicKernel_d_2:: Isolate_y_2	A model of *AlgebraicKernel_d_2::IsolateY_2*.
AlgebraicKernel_d_2:: Is_square_free_2	A model of *AlgebraicKernel_d_2::IsSquareFree_2*.
AlgebraicKernel_d_2:: Make_square_free_2	A model of *AlgebraicKernel_d_2::MakeSquareFree_2*.

AlgebraicKernel_d_2:: Square_free_factorize_2

A model of *AlgebraicKernel_d_2::SquareFreeFactorize_2*.

AlgebraicKernel_d_2:: Is_coprime_2	A model of *AlgebraicKernel_d_2::IsCoprime_2*.
AlgebraicKernel_d_2:: Make_coprime_2	A model of *AlgebraicKernel_d_2::MakeCoprime_2*.
AlgebraicKernel_d_2:: Solve_2	A model of *AlgebraicKernel_d_2::Solve_2*.
AlgebraicKernel_d_2:: Number_of_solutions_2	A model of *AlgebraicKernel_d_2::NumberOfSolutions_2*.
AlgebraicKernel_d_2:: Sign_at_2	A model of *AlgebraicKernel_d_2::SignAt_2*.
AlgebraicKernel_d_2:: Compare_x_2	A model of *AlgebraicKernel_d_2::CompareX_2*.
AlgebraicKernel_d_2:: Compare_y_2	A model of *AlgebraicKernel_d_2::CompareY_2*.
AlgebraicKernel_d_2:: Compare_xy_2	A model of *AlgebraicKernel_d_2::CompareXY_2*.
AlgebraicKernel_d_2:: Bound_between_x_2	A model of *AlgebraicKernel_d_2::BoundBetweenX_2*.
AlgebraicKernel_d_2:: Bound_between_y_2	A model of *AlgebraicKernel_d_2::BoundBetweenY_2*.

AlgebraicKernel_d_2:: Approximate_absolute_x_2

A model of *AlgebraicKernel_d_2::ApproximateAbsoluteX_2*.

AlgebraicKernel_d_2:: Approximate_absolute_y_2

A model of *AlgebraicKernel_d_2::ApproximateAbsoluteY_2*.

AlgebraicKernel_d_2:: Approximate_relative_x_2

A model of *AlgebraicKernel_d_2::ApproximateRelativeX_2*.

AlgebraicKernel_d_2:: Approximate_relative_y_2

A model of *AlgebraicKernel_d_2::ApproximateRelativeY_2*.

Operations

For each of the function objects above, there must exist a member function that requires no arguments and returns an instance of that function object. The name of the member function is the uncapitalized name of the type returned with the suffix *_object* appended. For example, for the function object *AlgebraicKernel_d_2::Bound_betweenX_2* the following member function must exist:

AlgebraicKernel_d_2::Bound_between_x_2

ak_2.bound_between_x_2_object() const

See Also

AlgebraicKernel_d_1 . . . . page 398

Functor
Concept

AlgebraicKernel_d_2::ConstructAlgebraicReal_2

Definition

Constructs an *AlgebraicKernel_d_2::Algebraic_real_2*.

Refines

AdaptableFunctor

Types

typedef AlgebraicKernel_d_2::Algebraic_real_2 *result_type;*

Operations

result_type *fo(int x, int y)* introduces an *AlgebraicKernel_d_2::Algebraic_real_2* initialized to (x, y).

result_type *fo(AlgebraicKernel_d_2::Bound x, AlgebraicKernel_d_2::Bound y)*

introduces an *AlgebraicKernel_d_2::Algebraic_real_2* initialized to (x, y).

result_type *fo(AlgebraicKernel_d_2::Coefficient x, AlgebraicKernel_d_2::Coefficient y)*

introduces an *AlgebraicKernel_d_2::Algebraic_real_2* initialized to (x, y).

result_type *fo(AlgebraicKernel_d_2::Algebraic_real_1 x, AlgebraicKernel_d_2::Algebraic_real_1 y)*

introduces an *AlgebraicKernel_d_2::Algebraic_real_2* initialized to (x, y).

result_type *fo.operator()(AlgebraicKernel_d_2::Polynomial_2 f,*
AlgebraicKernel_d_2::Polynomial_2 g,
AlgebraicKernel_d_2::size_type i)

introduces an *AlgebraicKernel_d_2::Algebraic_real_2* initialized to the i-th real common solution of f and g, with respect to xy-lexicographic order. The index starts at 0, that is, the system must have at least $i+1$ real solutions.
Precondition: f is square free.
Precondition: g is square free.
Precondition: f and g are coprime.

result_type *fo.operator()(AlgebraicKernel_d_2::Polynomial_2 f,*
AlgebraicKernel_d_2::Polynomial_2 g,
AlgebraicKernel_d_2::Bound x_l,
AlgebraicKernel_d_2::Bound x_u,
AlgebraicKernel_d_2::Bound y_l,
AlgebraicKernel_d_2::Bound y_u)

introduces an *AlgebraicKernel_d_2::Algebraic_real_2* initialized to the only real intersection of f and g in the open box $B = (x_l, x_u) \times (y_l, y_u)$.
Precondition: $x_l < x_u$
Precondition: $y_l < y_u$
Precondition: f is square free.
Precondition: g is square free.
Precondition: f and g are coprime.
Precondition: f and g have exactly one common solution in B
Precondition: f and g have no common solution on ∂B

See Also

AlgebraicKernel_d_1::ConstructAlgebraicReal_1 . . . . page 400

Functor
Concept

AlgebraicKernel_d_2::ComputePolynomialX_2

Definition

Computes a univariate square free polynomial p, such that the first coordinate of a given *AlgebraicKernel_d_2::Algebraic_real_2* is a real root of p.

Refines

AdaptableUnaryFunction

Types

typedef AlgebraicKernel_d_2::Polynomial_1	*result_type;*
typedef AlgebraicKernel_d_2::Algebraic_real_2	*argument_type;*

Operations

result_type *fo(argument_type a)*

Computes a univariate square free polynomial p, such that the first coordinate of a is a real root of p.

See Also

AlgebraicKernel_d_2::ComputePolynomialY_2 . . . . page 427

AlgebraicKernel_d_2::ComputePolynomialY_2

Definition

Computes a univariate square free polynomial p, such that the second coordinate of a given *AlgebraicKernel_d_2::Algebraic_real_2* is a real root of p.

Refines

AdaptableUnaryFunction

Types

typedef AlgebraicKernel_d_2::Polynomial_1	*result_type;*
typedef AlgebraicKernel_d_2::Algebraic_real_2	*argument_type;*

Operations

result_type *fo(argument_type a)*

Computes a univariate square free polynomial p, such that the second coordinate of a is a real root of p.

See Also

AlgebraicKernel_d_2::ComputePolynomialX_2 . . . . page 426

Functor Concept

AlgebraicKernel_d_2::Isolate_2

Definition

Computes an isolating box for a given *AlgebraicKernel_d_2::Algebraic_real_2*.

Refines

AdaptableFunctor

Types

typedef CGAL::cpp0x::array<AlgebraicKernel_d_1::Bound, 4> *result_type;*

Operations

result_type *fo(AlgebraicKernel_d_2::Algebraic_real_2 a, AlgebraicKernel_d_2::Polynomial_2 f)*

The returned *CGAL::cpp0x::array* $[xl, xu, yl, yu]$ represents an open isolating box $B = (xl, xu) \times (yl, yu)$ for a with respect to f.
Precondition: $f(a) \neq 0$
Postcondition: $a \in B$.
Postcondition: $\{r|f(r) = 0\} \cap \overline{B} = \emptyset$.

result_type *fo.operator()(AlgebraicKernel_d_2::Algebraic_real_2 a, AlgebraicKernel_d_2::Polynomial_2 f, AlgebraicKernel_d_2::Polynomial_2 g)*

The returned *CGAL::cpp0x::array* $[xl, xu, yl, yu]$ represents an open isolating box $B = (xl, xu) \times (yl, yu)$ for a with respect to the common solutions of f and g. It is not necessary that a is a common solution of f and g.
Postcondition: $a \in B$.
Postcondition: $\{r|f(r) = g(r) = 0\} \cap \overline{B} \in \{\{a\}, \emptyset\}$.

See Also

AlgebraicKernel_d_2::IsolateX_2 .... page 429
AlgebraicKernel_d_2::IsolateY_2 .... page 430
AlgebraicKernel_d_2::ComputePolynomialX_2 .... page 426
AlgebraicKernel_d_2::ComputePolynomialY_2 .... page 427

AlgebraicKernel_d_2::IsolateX_2

Definition

Computes an isolating interval for the first coordinate of an *AlgebraicKernel_d_2::Algebraic_real_2* with respect to the real roots of a univariate polynomial.

Refines

AdaptableBinaryFunction

Types

typedef std::pair<AlgebraicKernel_d_2::Bound,AlgebraicKernel_d_2::Bound> *result_type;*

typedef AlgebraicKernel_d_2::Algebraic_real_2 *first_argument_type;*

typedef AlgebraicKernel_d_2::Polynomial_1 *second_argument_type;*

Operations

result_type *fo(first_argument_type a, second_argument_type p)*

Computes an open isolating interval $I = (l,u)$ for the first coordinate x of a with respect to the real roots of p. It is not required that x is a root of p.
Postcondition: $x \in I$.
Postcondition: $p(\alpha) \neq 0 | \forall \alpha \in \bar{I} \backslash x$.

See Also

AlgebraicKernel_d_2::IsolateY_2 .... page 430
AlgebraicKernel_d_2::ComputePolynomialX_2 .... page 426
AlgebraicKernel_d_2::ComputePolynomialY_2 .... page 427

AlgebraicKernel_d_2::IsolateY_2

Definition

Computes an isolating interval for the second coordinate of an *AlgebraicKernel_d_2::Algebraic_real_2* with respect to the real roots of a univariate polynomial.

Refines

AdaptableBinaryFunction

Types

typedef std::pair<AlgebraicKernel_d_2::Bound,AlgebraicKernel_d_2::Bound> *result_type;*

typedef AlgebraicKernel_d_2::Algebraic_real_2 *first_argument_type;*

typedef AlgebraicKernel_d_2::Polynomial_1 *second_argument_type;*

Operations

result_type *fo(first_argument_type a, second_argument_type p)*

Computes an open isolating interval $I = (l, u)$ for the second coordinate y of a with respect to the real roots of p. It is not required that x is a root of p.
Postcondition: $y \in I$.
Postcondition: $p(\alpha) \neq 0 | \forall \alpha \in \bar{I} \backslash y$.

See Also

AlgebraicKernel_d_2::IsolateX_2 ... page 429
AlgebraicKernel_d_2::ComputePolynomialX_2 ... page 426
AlgebraicKernel_d_2::ComputePolynomialY_2 ... page 427

AlgebraicKernel_d_2::IsSquareFree_2

Definition

Computes whether the given bivariate polynomial is square free.

Refines

AdaptableUnaryFunction

Types

typedef bool	*result_type;*
typedef AlgebraicKernel_d_2::Polynomial_2	*argument_type;*

Operations

result_type   *fo(argument_type p)*

Computes whether p is square free.

See Also

AlgebraicKernel_d_2::MakeSquareFree_2 . . . . page 432
AlgebraicKernel_d_2::SquareFreeFactorize_2 . . . . page 433

AlgebraicKernel_d_2::MakeSquareFree_2

Definition

Returns a square free part of a bivariate polynomial.

Refines

AdaptableUnaryFunction

Types

typedef AlgebraicKernel_d_2::Polynomial_2	*result_type;*
typedef AlgebraicKernel_d_2::Polynomial_2	*argument_type;*

Operations

result_type *fo(argument_type p)*

Returns a square free part of p.

See Also

AlgebraicKernel_d_2::IsSquareFree_2 . . . . . . page 431
AlgebraicKernel_d_2::SquareFreeFactorize_2 . . . . . . page 433

AlgebraicKernel_d_2::SquareFreeFactorize_2

Definition

Computes a square free factorization of an *AlgebraicKernel_d_2::Polynomial_2*.

A polynomial p is factored into square free and pairwise coprime non-constant factors q_i with multiplicities m_i and a constant factor c, such that $p = c \cdot q_1^{m_1} \cdot \ldots \cdot q_n^{m_n}$.

The factor multiplicity pairs $< q_i, m_i >$ are written to the given output iterator. The constant factor c is not computed.

Refines

Assignable
CopyConstructible

Operations

template < class OutputIterator >
OutputIterator *fo(AlgebraicKernel_d_2::Polynomial_2 p, OutputIterator res)*

Copies in the output iterator the factors of a square free factorization of p, with their multiplicity, as objects of type *std::pair<AlgebraicKernel_d_2::Polynomial_2, AlgebraicKernel_d_2::Multiplicity_type>*.

See Also

AlgebraicKernel_d_2::IsSquareFree_2 .... page 431
AlgebraicKernel_d_2::MakeSquareFree_2 .... page 432

Functor
Concept

AlgebraicKernel_d_2::IsCoprime_2

Definition

Computes whether a given pair of bivariate polynomials is coprime.

Refines

AdaptableBinaryFunction

Types

typedef bool	*result_type;*
typedef AlgebraicKernel_d_2::Polynomial_2	*first_argument_type;*
typedef AlgebraicKernel_d_2::Polynomial_2	*second_argument_type;*

Operations

result_type *fo(first_argument_type p1, second_argument_type p2)*

Computes whether f and g are coprime.

See Also

AlgebraicKernel_d_2::MakeCoprime_2 . . . . page 435

AlgebraicKernel_d_2::MakeCoprime_2

Functor
Concept

Definition

Computes for a given pair of bivariate polynomials p_1, p_2 their common part g and coprime parts q_1, q_2 respectively.

That is, it computes g, q_1, q_2 such that:
$c_1 \cdot p_1 = g \cdot q_1$ for some constant c_1 and
$c_2 \cdot p_2 = g \cdot q_2$ for some constant c_2, such that q_1 and q_2 are coprime.

Refines

AdaptableFunctor with five arguments

Types

typedef bool *result_type;*

Operations

result_type *fo.operator()(AlgebraicKernel_d_2::Polynomial_2 p1,*
AlgebraicKernel_d_2::Polynomial_2 p2,
AlgebraicKernel_d_2::Polynomial_2 & g,
AlgebraicKernel_d_2::Polynomial_2 & q1,
AlgebraicKernel_d_2::Polynomial_2 & q2)

Computes g, q_1, q_2 as described above.
Returns whether p_1 and p_2 where already coprime.

See Also

AlgebraicKernel_d_2::IsCoprime_2 . . . . page 434

AlgebraicKernel_d_2::Solve_2

Definition

Computes the real zero-dimensional solutions of a bivariate polynomial system. The multiplicity stored in the output iterator is the multiplicity in the system.

Refines

Assignable CopyConstructible

Operations

A model of this type must provide:

template < class OutputIterator >
OutputIterator *fo.operator()(AlgebraicKernel_d_2::Polynomial_2 f, AlgebraicKernel_d_2::Polynomial_2 g, OutputIterator res)*

Computes all common solutions of f and g with multiplicity, and copies them as objects of type *std::pair<AlgebraicKernel_d_2::Algebraic_real_2, AlgebraicKernel_d_2::Multiplicity_type>* in *res*.
Precondition: f is square free.
Precondition: g is square free.
Precondition: f and g are coprime.

template < class OutputIterator >
OutputIterator *fo.operator()(AlgebraicKernel_d_2::Polynomial_2 f, AlgebraicKernel_d_2::Polynomial_2 g, AlgebraicKernel_d_2::Bound xl, AlgebraicKernel_d_2::Bound xu, AlgebraicKernel_d_2::Bound yl, AlgebraicKernel_d_2::Bound yu, OutputIterator res)*

Computes all common solutions of f and g in the closed box $[xl,xu] \times [yl,yu]$, and copies them as objects of type *std::pair<AlgebraicKernel_d_2::Algebraic_real_2, AlgebraicKernel_d_2::Multiplicity_type>* in *res*.
Precondition: f is square free.
Precondition: g is square free.
Precondition: f and g are coprime.

AlgebraicKernel_d_2::NumberOfSolutions_2

Functor
Concept

Definition

Computes the number of real solutions of the given bivariate polynomial system.

Refines

AdaptableBinaryFunction

Types

typedef AlgebraicKernel_d_2::size_type	*result_type;*
typedef AlgebraicKernel_d_2::Polynomial_2	*first_argument_type;*
typedef AlgebraicKernel_d_2::Polynomial_2	*second_argument_type;*

A model of this type must provide:

Operations

result_type *fo(first_argument_type f, second_argument_type g)*

Returns the number of real solutions of the bivariate polynomial system (f,g).
Precondition: f is square free.
Precondition: g is square free.
Precondition: f and g are coprime.

See Also

AlgebraicKernel_d_2::ConstructAlgebraicReal_2 . . . . page 424

AlgebraicKernel_d_2::SignAt_2

Definition

Computes the sign of a bivariate polynomial *AlgebraicKernel_d_2::Polynomial_2* at a value of type *AlgebraicKernel_d_2::Algebraic_real_2*.

Refines

AdaptableBinaryFunction

Types

AlgebraicKernel_d_2::SignAt_2:: result_type	Type convertible to *CGAL::Sign*
typedef AlgebraicKernel_d_2::Polynomial_2	*first_argument_type;*
typedef AlgebraicKernel_d_2::Algebraic_real_2	*second_argument_type;*

Operations

result_type *fo(first_argument_type p, second_argument_type a)*

Computes the sign of a bivariate polynomial p evaluated at a.

See Also

AlgebraicKernel_d_2::IsZeroAt_2 . . . . page 439
AlgebraicKernel_d_1::SignAt_1 . . . . page 412

AlgebraicKernel_d_2::IsZeroAt_2

Functor
Concept

Definition

Computes whether an *AlgebraicKernel_d_2::Polynomial_2* is zero at a given *AlgebraicKernel_d_2::Algebraic_real_2*.

Refines

AdaptableBinaryFunction

Types

AlgebraicKernel_d_2::IsZeroAt_2:: result_type

Type convertible to *bool*

typedef AlgebraicKernel_d_2::Polynomial_2 *first_argument_type;*
typedef AlgebraicKernel_d_2::Algebraic_real_2 *second_argument_type;*

Operations

result_type *fo(first_argument_type p, second_argument_type a)*

Computes whether p is zero at a.

See Also

AlgebraicKernel_d_2::SignAt_2 . . . . page 438
AlgebraicKernel_d_1::IsZeroAt_1 . . . . page 413

AlgebraicKernel_d_2::ComputeX_2

Definition

Computes the first coordinate of an *AlgebraicKernel_d_2::AlgebraicReal_2*.

Refines

AdaptableUnaryFunction

Types

typedef AlgebraicKernel_d_2::Algebraic_real_1	*result_type;*
typedef AlgebraicKernel_d_2::Algebraic_real_2	*argument_type;*

Operations

A model of this type must provide:

result_type *fo(argument_type a)*

Computes the first coordinate of *a*.

See Also

AlgebraicKernel_d_2::ComputeY_2 .... page 441

AlgebraicKernel_d_2::ComputeY_2

Functor
Concept

Definition

Computes the second coordinate of an *AlgebraicKernel_d_2::AlgebraicReal_2*.

Refines

AdaptableUnaryFunction

Types

typedef AlgebraicKernel_d_2::Algebraic_real_1	*result_type;*
typedef AlgebraicKernel_d_2::Algebraic_real_2	*argument_type;*

Operations

A model of this type must provide:

result_type *fo(argument_type a)*

Computes the second coordinate of *a*.

See Also

AlgebraicKernel_d_2::ComputeY_2 . . . . page 441

Functor Concept

AlgebraicKernel_d_2::CompareX_2

Definition

Compares the first coordinates of *AlgebraicKernel_d_2::Algebraic_real_2*s.

Refines

AdaptableBinaryFunction

Types

AlgebraicKernel_d_2::CompareX_2:: result_type

Type convertible to *CGAL::Comparison_result*

typedef AlgebraicKernel_d_2::Algebraic_real_2 *first_argument_type;*
typedef AlgebraicKernel_d_2::Algebraic_real_2 *second_argument_type;*

Operations

The following operators and their symmetric counterparts are required:

result_type *fo(first_argument_type a, second_argument_type b)*

Compares the first coordinates of *a* and *b*.

result_type *fo(AlgebraicKernel_d_2::Algebraic_real_2 a, int x)*

Compares the first coordinate of *a* with *x*.

result_type *fo(AlgebraicKernel_d_2::Algebraic_real_2 a, AlgebraicKernel_d_2::Bound x)*

Compares the first coordinate of *a* with *x*.

result_type *fo(AlgebraicKernel_d_2::Algebraic_real_2 a, AlgebraicKernel_d_2::Coefficient x)*

Compares the first coordinate of *a* with *x*.

result_type *fo(AlgebraicKernel_d_2::Algebraic_real_2 a, AlgebraicKernel_d_2::Algebraic_real_1 x)*

Compares the first coordinate of *a* with *x*.

See Also

AlgebraicKernel_d_2::CompareY_2 . . . . page 443
AlgebraicKernel_d_2::CompareXY_2 . . . . page 444

AlgebraicKernel_d_2::CompareY_2

Functor
Concept

Definition

Compares the second coordinated of *AlgebraicKernel_d_2::Algebraic_real_2*s.

Refines

AdaptableBinaryFunction

Types

AlgebraicKernel_d_2::CompareY_2:: result_type

Type convertible to *CGAL::Comparison_result*

typedef AlgebraicKernel_d_2::Algebraic_real_2 *first_argument_type;*
typedef AlgebraicKernel_d_2::Algebraic_real_2 *second_argument_type;*

Operations

The following operators and their symmetric counterparts are required:

result_type *fo(first_argument_type a, second_argument_type b)*

Compares the second coordinates of *a* and *b*.

result_type *fo(AlgebraicKernel_d_2::Algebraic_real_2 a, int y)*

Compares the second coordinate of *a* with *y*.

result_type *fo(AlgebraicKernel_d_2::Algebraic_real_2 a, AlgebraicKernel_d_2::Bound y)*

Compares the second coordinate of *a* with *y*.

result_type *fo(AlgebraicKernel_d_2::Algebraic_real_2 a, AlgebraicKernel_d_2::Coefficient y)*

Compares the second coordinate of *a* with *y*.

result_type *fo(AlgebraicKernel_d_2::Algebraic_real_2 a, AlgebraicKernel_d_2::Algebraic_real_1 y)*

Compares the second coordinate of *a* with *y*.

See Also

AlgebraicKernel_d_2::CompareX_2 . . . . page 442
AlgebraicKernel_d_2::CompareXY_2 . . . . page 444

Functor Concept

AlgebraicKernel_d_2::CompareXY_2

Definition

Compares *AlgebraicKernel_d_2::Algebraic_real_2*s lexicographically.

Refines

AdaptableBinaryFunction

Types

AlgebraicKernel_d_2::CompareXY_2:: result_type

Type convertible to *CGAL::Comparison_result*

typedef AlgebraicKernel_d_2::Algebraic_real_2 first_argument_type;
typedef AlgebraicKernel_d_2::Algebraic_real_2 second_argument_type;

Operations

result_type fo(first_argument_type a, second_argument_type b)

Compares a and b lexicographically.

result_type fo(AlgebraicKernel_d_2::Algebraic_real_2 a, int x, int y)

Compares a with (x,y) lexicographically.

result_type fo.operator()(AlgebraicKernel_d_2::Algebraic_real_2 a, AlgebraicKernel_d_2::Bound x, AlgebraicKernel_d_2::Bound y)

Compares a with (x,y) lexicographically.

result_type fo.operator()(AlgebraicKernel_d_2::Algebraic_real_2 a, AlgebraicKernel_d_2::Coefficient x, AlgebraicKernel_d_2::Coefficient y)

Compares a with (x,y) lexicographically.

See Also

AlgebraicKernel_d_2::CompareX_2 . . . . page 442
AlgebraicKernel_d_2::CompareY_2 . . . . page 443

AlgebraicKernel_d_2::ApproximateAbsoluteX_2

Functor
Concept

Definition

A model of AlgebraicKernel_d_2::ApproximateAbsoluteX_2 is an *AdaptableBinaryFunction* that computes an approximation of the x-coordinate of an *AlgebraicKernel_d_2::Algebraic_real_2* value with respect to a given absolute precision.

Refines

AdaptableBinaryFunction

Types

typedef std::pair<AlgebraicKernel_d_1::Bound, AlgebraicKernel_d_1::Bound>

	result_type;
typedef AlgebraicKernel_d_2::Algebraic_real_2	*first_argument_type;*
typedef int	*second_argument_type;*

Operations

result_type *fo(first_argument_type v, second_argument_type a)*

The function computes a pair p of *AlgebraicKernel_d_1::Bound*, where $p.first$ represents the lower approximation and $p.second$ represents the upper approximation. The pair p approximates the x-coordinate x of the *AlgebraicKernel_d_2::Algebraic_real_2* value v with respect to the absolute precision a.
Postcondition: $p.first <= x$
Postcondition: $x <= p.second$
Postcondition: $(x - p.first) <= 2^{-a}$
Postcondition: $(p.second - x) <= 2^{-a}$

See Also

AlgebraicKernel_d_2::ApproximateRelativeX_2 . . . . page 447
AlgebraicKernel_d_1::ApproximateAbsolute_1 . . . . page 416
AlgebraicKernel_d_1::ApproximateRelative_1 . . . . page 417

Functor Concept

AlgebraicKernel_d_2::ApproximateAbsoluteY_2

Definition

A model of AlgebraicKernel_d_2::ApproximateAbsoluteY_2 is an *AdaptableBinaryFunction* that computes an approximation of the y-coordinate of an *AlgebraicKernel_d_2::Algebraic_real_2* value with respect to a given absolute precision.

Refines

AdaptableBinaryFunction

Types

typedef std::pair<AlgebraicKernel_d_1::Bound, AlgebraicKernel_d_1::Bound>

result_type;

typedef AlgebraicKernel_d_2::Algebraic_real_2 *first_argument_type;*

typedef int *second_argument_type;*

Operations

result_type *fo(first_argument_type v, second_argument_type a)*

The function computes a pair p of *AlgebraicKernel_d_1::Bound*, where $p.first$ represents the lower approximation and $p.second$ represents the upper approximation. The pair p approximates the y-coordinate y of the *AlgebraicKernel_d_2::Algebraic_real_2* value v with respect to the absolute precision a.
Postcondition: $p.first <= y$
Postcondition: $y <= p.second$
Postcondition: $(y - p.first) <= 2^{-a}$
Postcondition: $(p.second - y) <= 2^{-a}$

See Also

AlgebraicKernel_d_2::ApproximateRelativeY_2 . . . . page 448
AlgebraicKernel_d_1::ApproximateAbsolute_1 . . . . page 416
AlgebraicKernel_d_1::ApproximateRelative_1 . . . . page 417

AlgebraicKernel_d_2::ApproximateRelativeX_2

Definition

A model of AlgebraicKernel_d_2::ApproximateRelativeX_2 is an *AdaptableBinaryFunction* that computes an approximation of the x-coordinate of an *AlgebraicKernel_d_2::Algebraic_real_2* value with respect to a given relative precision.

Refines

AdaptableBinaryFunction

Types

typedef std::pair<AlgebraicKernel_d_1::Bound, AlgebraicKernel_d_1::Bound>

	result_type;
typedef AlgebraicKernel_d_2::Algebraic_real_2	*first_argument_type;*
typedef int	*second_argument_type;*

Operations

result_type *fo(first_argument_type v, second_argument_type a)*

The function computes a pair p of *AlgebraicKernel_d_1::Bound*, where $p.first$ represents the lower approximation and $p.second$ represents the upper approximation. The pair p approximates the x-coordinate x of the *AlgebraicKernel_d_2::Algebraic_real_2* value v with respect to the relative precision a.
Postcondition: $p.first <= x$
Postcondition: $x <= p.second$
Postcondition: $(x - p.first) <= 2^{-a}|x|$
Postcondition: $(p.second - x) <= 2^{-a}|x|$

See Also

AlgebraicKernel_d_2::ApproximateAbsoluteY_2 . . . . . . page 446
AlgebraicKernel_d_1::ApproximateAbsolute_1 . . . . . . page 416
AlgebraicKernel_d_1::ApproximateRelative_1 . . . . . . page 417

AlgebraicKernel_d_2::ApproximateRelativeY_2

Definition

A model of AlgebraicKernel_d_2::ApproximateRelativeY_2 is an *AdaptableBinaryFunction* that computes an approximation of the y-coordinate of an *AlgebraicKernel_d_2::Algebraic_real_2* value with respect to a given relative precision.

Refines

AdaptableBinaryFunction

Types

typedef std::pair<AlgebraicKernel_d_1::Bound, AlgebraicKernel_d_1::Bound>	
	result_type;
typedef AlgebraicKernel_d_2::Algebraic_real_2	*first_argument_type;*
typedef int	*second_argument_type;*

Operations

result_type *fo(first_argument_type v, second_argument_type a)*

The function computes a pair p of *AlgebraicKernel_d_1::Bound*, where $p.first$ represents the lower approximation and $p.second$ represents the upper approximation. The pair p approximates the y-coordinate y of the *AlgebraicKernel_d_2::Algebraic_real_2* value v with respect to the relative precision a.
Postcondition: $p.first <= y$
Postcondition: $y <= p.second$
Postcondition: $(y - p.first) <= 2^{-a}|y|$
Postcondition: $(p.second - y) <= 2^{-a}|y|$

See Also

AlgebraicKernel_d_2::ApproximateAbsoluteY_2 . . . . . page 446
AlgebraicKernel_d_1::ApproximateAbsolute_1 . . . . . page 416
AlgebraicKernel_d_1::ApproximateRelative_1 . . . . . page 417

AlgebraicKernel_d_2::BoundBetweenX_2

Functor Concept

Definition

Computes a number of type *AlgebraicKernel_d_1::Bound* in-between the first coordinates of two *AlgebraicKernel_d_2::AlgebraicReal_2*.

Refines

AdaptableBinaryFunction

Types

typedef AlgebraicKernel_d_1::Bound	*result_type;*
typedef AlgebraicKernel_d_2::Algebraic_real_2	*first_argument_type;*
typedef AlgebraicKernel_d_2::Algebraic_real_2	*second_argument_type;*

Operations

result_type *fo(first_argument_type a, second_argument_type b)*

Computes a number of type *AlgebraicKernel_d_1::Bound* in-between the first coordinates of *a* and *b*.
Precondition: $a_x \neq b_x$.

See Also

AlgebraicKernel_d_2::BoundBetweenY_2 . . . . page 450

Functor Concept

AlgebraicKernel_d_2::BoundBetweenY_2

Definition

Computes a number of type *AlgebraicKernel_d_1::Bound* in-between the second coordinates of two *AlgebraicKernel_d_2::AlgebraicReal_2*.

Refines

AdaptableBinaryFunction

Types

typedef AlgebraicKernel_d_1::Bound	*result_type;*
typedef AlgebraicKernel_d_2::Algebraic_real_2	*first_argument_type;*
typedef AlgebraicKernel_d_2::Algebraic_real_2	*second_argument_type;*

Operations

result_type *fo(first_argument_type a, second_argument_type b)*

Computes a number of type *AlgebraicKernel_d_1::Bound* in-between the second coordinates of *a* and *b*.
Precondition: $a_y \neq b_y$.

See Also

AlgebraicKernel_d_2::BoundBetweenX_2 . . . . page 449

CGAL::Algebraic_kernel_d_2<Coeff>

#include <CGAL/Algebraic_kernel_d_2.h>

Definition

This class is based on an algorithm computing a geometric-topological analysis of a single curve [EKW07] and of a pair of curves [EK08]. The main idea behind both analyses is to compute the critical x-coordinates of curves and curve pairs by projection (resultants), and compute additional information about the critical fibers using subresultants and Sturm-Habicht sequences [GVRLR98]. With that information, the fiber at critical x-coordinates is computed by a variant of the Bitstream Descartes method. See also [Ker09] for a comprehensive description of these techniques.

A point p of type *Algebraic_real_2* is represented by its x-coordinate x_0 (as described in the *Algebraic_kernel_d_1* paragraph above), an algebraic curve where p lies on, and an integer i, denoting that p is the ith point in the fiber at x_0, counted from the bottom (ignoring a possible vertical line at x_0). This determines the point uniquely, but the y-coordinate is not stored internally in terms of an *Algebraic_real_1* object. Querying such a representation by calling *Compute_y_2* is a time-consuming step, and should be avoided for efficiency reasons if possible. Note that this representation is not exposed in the interface.

The template argument *Coeff* determines the coefficient type of the kernel, which is also the innermost coefficient type of the supported polynomials.

Currently, the following coefficient types are supported:
– *Gmpz*, *Gmpq*, (requires configuration with external libraries GMP, MPFR and MPFI)
– *CORE::BigInt*, *CORE::BigRat*, (requires configuration with external library GMP)
– *leda_integer*, *leda_rational*. (requires configuration with external library LEDA)

advanced

The template argument type can also be set to *Sqrt_extension<NT,ROOT>*, where *NT* is one of the types listed above. *ROOT* should be one of the integer types. See also the documentation of *Sqrt_extension<NT,ROOT>*.

advanced

Is Model for the Concepts

AlgebraicKernel_d_2.

Types

Algebraic_kernel_d_2<Coeff>:: Coefficient	Same type as the template argument *Coeff*.
Algebraic_kernel_d_2<Coeff>:: Polynomial_2	A model of *AlgebraicKernel_d_2::Polynomial_2*

.

Algebraic_kernel_d_2<Coeff>:: Algebraic_real_2

A model of *AlgebraicKernel_d_2::AlgebraicReal_2*

Algebraic_kernel_d_2<Coeff>:: Bound

The choice of *Coeff* also determines the provided bound, type. In case of *Coeff* is - *Gmpz* or *Gmpq* this is *Gmpq*
- *CORE::BigInt* or *CORE::BigInt* this is *CORE::BigRat*
- *leda_integer* or *leda_integer* this is *leda_rational*

Algebraic_kernel_d_2<Coeff>:: Multiplicity_type

The multiplicity type is *int*.

See Also

AlgebraicKernel_d_1 . . . . page 398
AlgebraicKernel_d_2 . . . . page 422
Polynomial_d . . . . page 261
CGAL::Algebraic_kernel_d_2<Coeff> . . . . page 451

Part III

Combinatorial Algorithms

Chapter 9

Monotone and Sorted Matrix Search

Michael Hoffmann

CGAL::monotone_matrix_search and *CGAL::sorted_matrix_search* are techniques that deal with the problem of efficiently finding largest entries in matrices with certain structural properties. Many concrete problems can be modelled as matrix search problems, and for some of them we provide explicit solutions that allow you to solve them without knowing about the matrix search technique. Examples are, the computation of all furthest neighbors for the vertices of a convex polygon, maximal k-gons inscribed into a planar point set, and computing rectangular p-centers.

Monotone and Sorted Matrix Search Reference Manual

Michael Hoffmann

This chapter describes concepts, classes, and functions for monotone and sorted matrix search.

Assertions

The optimization code uses infix *OPTIMISATION* in the assertions, e.g. defining the compiler flag *CGAL_OPTIMISATION_NO_PRECONDITIONS* switches precondition checking off, cf. Section 2.8.

9.1 Classified References Pages

9.2 Alphabetical List of Reference Pages

Function

CGAL::monotone_matrix_search

advanced

Definition

The function *monotone_matrix_search* computes the maxima for all rows of a totally monotone matrix.

More precisely, monotony for matrices is defined as follows.

Let K be a totally ordered set, $M \in K^{(n,m)}$ a matrix over K and for $0 \leq i < n$:

$$rmax_M(i) :\in \left\{ \min_{0 \leq j < m} j \;\middle|\; M[i,j] = \max_{0 \leq k < m} M[i,k] \right\}$$

the (leftmost) column containing the maximum entry in row i. M is called monotone, iff

$$\forall 0 \leq i_1 < i_2 < n \,:\, rmax_M(i_1) \leq rmax_M(i_2)\,.$$

M is totally monotone, iff all of its submatrices are monotone (or equivalently: iff all 2×2 submatrices are monotone).

#include <CGAL/monotone_matrix_search.h>

```
template < class Matrix, class RandomAccessIC, class Compare_strictly >
void    monotone_matrix_search(
            Matrix m,
            RandomAccessIC t,
            Compare_strictly compare_strictly = less< Matrix::Value >())
```

computes the maximum (as specified by *compare_strictly*) entry for each row of m and writes the corresponding column to t, i.e. *t[i]* is set to the index of the column containing the maximum element in row i. The maximum m_r of a row r is the leftmost element for which $compare\_strictly(m_r, x)$ is false for all elements x in r.

Precondition: t points to a structure of size at least *m.number_of_rows()*

Requirement:

1. *Matrix* is a model for *MonotoneMatrixSearchTraits*.
2. Value type of *RandomAccessIC* is *int*.
3. If *compare_strictly* is defined, it is an adaptable binary function: *Matrix::Value* $\times$ *Matrix::Value* $\rightarrow$ *bool* describing a strict (non-reflexive) total ordering on *Matrix::Value*.

See Also

MonotoneMatrixSearchTraits . . . page 462
CGAL::all_furthest_neighbors_2 . . . page 3933
CGAL::maximum_area_inscribed_k_gon_2 . . . page 3912
CGAL::maximum_perimeter_inscribed_k_gon_2 . . . page 3914
CGAL::extremal_polygon_2 . . . page 3916

Implementation

The implementation uses an algorithm by Aggarwal et al.[AKM+87]. The runtime is linear in the number of rows and columns of the matrix.

advanced

Class

CGAL::Dynamic_matrix<M>

advanced

Definition

The class *Dynamic_matrix<M>* is an adaptor for an arbitrary matrix class *M* to provide the dynamic operations needed for monotone matrix search.

Requirements

M is a model for *BasicMatrix*.

#include <CGAL/Dynamic_matrix.h>

Is Model for the Concepts

MonotoneMatrixSearchTraits . . . . page 462
BasicMatrix . . . . page 464

Creation

Dynamic_matrix<M> d(M m);

initializes *d* to *m*. *m* is *not* copied, we only store a reference.

Operations

int *d.number_of_columns() const*

returns the number of columns.

int *d.number_of_rows() const*

returns the number of rows.

Entry *d(int row, int column)*

returns the entry at position (*row*, *column*).
Precondition:
$0 \leq row < number\_of\_rows()$ and
$0 \leq column < number\_of\_columns()$.

void *d.replace_column(int old, int new)*

replace column *old* with column number *new*.
Precondition:
$0 \leq old, new < number\_of\_columns()$.

*Matrix** *d.extract_all_even_rows() const*

returns a new Matrix consisting of all rows of d with even index, (i.e. first row is row 0 of d, second row is row 2 of d etc.).
Precondition: *number_of_rows()* > 0.

void *d.shrink_to_quadratic_size()*

deletes the rightmost columns, such that d becomes quadratic.
Precondition:
number_of_columns() $\geq$ *number_of_rows()*.
Postcondition:
number_of_rows() $==$ *number_of_columns()*.

See Also

CGAL::monotone_matrix_search . . . . page 458
MonotoneMatrixSearchTraits . . . . page 462
BasicMatrix . . . . page 464

Implementation

All operations take constant time except for *extract_all_even_rows* which needs time linear in the number of rows.

advanced

MonotoneMatrixSearchTraits

advanced

Definition

The concept MonotoneMatrixSearchTraits is a refinement of *BasicMatrix* and defines types and operations needed to compute the maxima for all rows of a totally monotone matrix using the function *monotone_matrix_search*.

Types

MonotoneMatrixSearchTraits:: Value The type of a matrix entry.

Operations

int *m.number_of_columns() const*

returns the number of columns.

int *m.number_of_rows() const*

returns the number of rows.

Entry *m.operator()(int row, int column) const*

returns the entry at position (*row*, *column*).
Precondition:
$0 \leq row < number\_of\_rows()$ and
$0 \leq column < number\_of\_columns()$.

void *m.replace_column(int old, int new)*

replace column *old* with column number *new*.
Precondition:
$0 \leq old, new < number\_of\_columns()$.

*Matrix** *m.extract_all_even_rows() const*

returns a new Matrix consisting of all rows of *m* with even index, (i.e. first row is row 0 of *m*, second row is row 2 of *m* etc.).
Precondition: $number\_of\_rows() > 0$.

void *m.shrink_to_quadratic_size()*

deletes the rightmost columns, such that *m* becomes quadratic.
Precondition:
number_of_columns() $\geq$ *number_of_rows()*.
Postcondition:
number_of_rows() == *number_of_columns()*.

Notes

- For the sake of efficiency (and in order to achieve the time bounds claimed for *monotone_matrix_search*), all these operations have to be realized in constant time – except for *extract_all_even_rows* which may take linear time.
- There is an adaptor *Dynamic_matrix* that can be used to add most of the functionality described above to arbitrary matrix classes.

Has Models

CGAL::Dynamic_matrix<M> . . . . page 460

See Also

CGAL::monotone_matrix_search . . . . page 458

advanced

Concept

BasicMatrix

advanced

Definition

A class has to provide the following types and operations in order to be a model for *BasicMatrix*.

Types

BasicMatrix:: Value — The type of a matrix entry. It has to define a copy constructor.

Operations

int *m.number_of_columns() const*

returns the number of columns.

int *m.number_of_rows() const*

returns the number of rows.

Entry *m.operator()(int row, int column) const*

returns the entry at position (*row*, *column*).
Precondition:
$0 \leq row < number\_of\_rows()$ and
$0 \leq column < number\_of\_columns()$.

Has Models

CGAL::Dynamic_matrix<M> . . . . page 460

See Also

MonotoneMatrixSearchTraits . . . . page 462
SortedMatrixSearchTraits . . . . page 470

advanced

CGAL::sorted_matrix_search

advanced

Definition

The function *sorted_matrix_search* selects the smallest entry in a set of sorted matrices that fulfills a certain feasibility criterion.

More exactly, a matrix $M = (m_{ij}) \in S^{r \times l}$ (over a totally ordered set S) is sorted, iff

$$\forall 1 \leq i \leq r,\ 1 \leq j < l\ :\ m_{ij} \leq m_{i(j+1)}\ \ \mathit{and}$$
$$\forall 1 \leq i < r,\ 1 \leq j \leq l\ :\ m_{ij} \leq m_{(i+1)j}\ \ .$$

Now let $\mathcal{M}$ be a set of n sorted matrices over S and f be a monotone predicate on S, i.e.

$$f\ :\ S \longrightarrow \mathit{bool} \quad \text{with} \quad f(r) \implies \forall t \in S, t > r\ :\ f(t)\ .$$

If we assume there is any feasible element in one of the matrices in $\mathcal{M}$, there certainly is a smallest such element. This is the one we are searching for.

The feasibility test as well as some other parameters can (and have to) be customized through a traits class.

#include <CGAL/sorted_matrix_search.h>

template < class RandomAccessIterator, class Traits >
Traits::Value *sorted_matrix_search(RandomAccessIterator f, RandomAccessIterator l, Traits t)*

returns the element x in one of the sorted matrices from the range $[f, l)$, for which *t.is_feasible(x)* is true and *t.compare(x, y)* is true for all other y values from any matrix for which *t.is_feasible(y)* is true.

Precondition:

1. All matrices in $[f, l)$ are sorted according to *Traits::compare_non_strictly*.
2. There is at least one entry x in a matrix $M \in [f, l)$ for which *Traits::is_feasible(x)* is true.

Requirement:

1. *Traits* is a model for *SortedMatrixSearchTraits*.
2. Value type of *RandomAccessIterator* is *Traits::Matrix*.

See Also

SortedMatrixSearchTraits . . . . page 470

Function

Implementation

The implementation uses an algorithm by Frederickson and Johnson[FJ83, FJ84] and runs in $O(n \cdot k + f \cdot \log(n \cdot k))$, where n is the number of input matrices, k denotes the maximal dimension of any input matrix and f the time needed for one feasibility test.

Example

In the following program we build a random vector $a = (a_i)_{i=1,\ldots,5}$ (elements drawn uniformly from $\{0,\ldots,99\}$) and construct a Cartesian matrix M containing as elements all sums $a_i + a_j$, $i, j \in \{1,\ldots,5\}$. If a is sorted, M is sorted as well. So we can apply *sorted_matrix_search* to compute the upper bound for the maximal entry of a in M.

```
#include <CGAL/Random.h>
#include <CGAL/Cartesian_matrix.h>
#include <CGAL/sorted_matrix_search.h>
#include <vector>
#include <algorithm>
#include <iterator>
#include <functional>

typedef int                                           Value;
typedef std::vector<Value>                            Vector;
typedef Vector::iterator                              Value_iterator;
typedef std::vector<Vector>                           Vector_cont;
typedef CGAL::Cartesian_matrix<std::plus<int>,
                               Value_iterator,
                               Value_iterator>  Matrix;

int main()
{
  // set of vectors the matrices are build from:
  Vector_cont vectors;

  // generate a random vector and sort it:
  Vector a;
  const int n = 5;
  for (int i = 0; i < n; ++i)
    a.push_back(CGAL::default_random(100));
  std::sort(a.begin(), a.end());
  std::cout << "a = ( ";
  std::copy(a.begin(), a.end(), std::ostream_iterator<int>(std::cout,"
   "));
  std::cout << ")\n";

  // build a Cartesian matrix from a:
  Matrix M(a.begin(), a.end(), a.begin(), a.end());

  // search for an upper bound for max(a):
  Value bound = a[n-1];
  Value upper_bound =
  CGAL::sorted_matrix_search(
    &M, &M + 1,
    CGAL::sorted_matrix_search_traits_adaptor(
```

```
      std::bind2nd(std::greater_equal<Value>(), bound), M));
  std::cout << "Upper bound for " << bound << " is "
            << upper_bound << "." << std::endl;

  return 0;
}
```

File: examples/Matrix_search/sorted_matrix_search.cpp

advanced

Class

CGAL::Sorted_matrix_search_traits_adaptor<F,M>

advanced

#include <CGAL/Sorted_matrix_search_traits_adaptor.h>

Definition

The class *Sorted_matrix_search_traits_adaptor<F,M>* can be used as an adaptor to create sorted matrix search traits classes for arbitrary feasibility test and matrix classes F resp. M.

Is Model for the Concepts

SortedMatrixSearchTraits . . . . page 470

Requirements

1. M is a model for *BasicMatrix and*
2. F defines a copy constructor and a monotone *bool operator()(const Value&)*.

Creation

Sorted_matrix_search_traits_adaptor<F,M> t(const F& m);

initializes t to use m for feasibility testing.

Types

Sorted_matrix_search_traits_adaptor<F,M>:: Matrix

typedef to M.

Sorted_matrix_search_traits_adaptor<F,M>:: Value

typedef to *Matrix::Value*.

Sorted_matrix_search_traits_adaptor<F,M>:: Compare_strictly

typedef to *std::less<Value>*.

Sorted_matrix_search_traits_adaptor<F,M>:: Compare_non_strictly

typedef to *std::less_equal<Value>*.

Operations

Compare_strictly *t.compare_strictly() const*

returns the *Compare_strictly* object to be used for the search.

Compare_non_strictly

t.compare_non_strictly() const

returns the *Compare_non_strictly* object to be used for the search.

bool *t.is_feasible(const Value& a)*

uses the feasibility test given during creation.

advanced

Concept

SortedMatrixSearchTraits

advanced

Definition

The concept SortedMatrixSearchTraits defines types and operations needed to compute the smallest entry in a set of sorted matrices that fulfills a certain feasibility criterion using the function *sorted_matrix_search*.

Types

SortedMatrixSearchTraits:: Matrix — The class used for representing matrices. It has to be a model for *BasicMatrix*.

typedef Matrix::Value

Value; — The class used for representing the matrix elements.

SortedMatrixSearchTraits:: Compare_strictly — An adaptable binary function class: $Value \times Value \rightarrow bool$ defining a non-reflexive total order on *Value*. This determines the direction of the search.

SortedMatrixSearchTraits:: Compare_non_strictly

An adaptable binary function class: $Value \times Value \rightarrow bool$ defining the reflexive total order on *Value* corresponding to *Compare_strictly*.

Operations

Compare_strictly *t.compare_strictly() const*

returns the *Compare_strictly* object to be used for the search.

Compare_non_strictly

t.compare_non_strictly() const

returns the *Compare_non_strictly* object to be used for the search.

bool *t.is_feasible(const Value& a)*

The predicate to determine whether an element *a* is feasible. It has to be monotone in the sense that *compare(a, b)* and *is_feasible(a)* imply *is_feasible(b)*.

Has Models

See Also

advanced

Chapter 10

Linear and Quadratic Programming Solver

Kaspar Fischer, Bernd Gärtner, Sven Schönherr, and Frans Wessendorp

Contents

10.1 Which Programs can be Solved?

This package lets you solve *convex quadratic programs* of the general form

$$\begin{array}{lll} \text{(QP)} & \text{minimize} & \mathbf{x}^T D\mathbf{x}+\mathbf{c}^T\mathbf{x}+c_0 \\ & \text{subject to} & A\mathbf{x} \gtreqless \mathbf{b}, \\ & & \mathbf{l}\leq \mathbf{x}\leq \mathbf{u} \end{array}$$

in n real variables $\mathbf{x}=(x_0,\ldots,x_{n-1})$. Here,

- A is an $m\times n$ matrix (the constraint matrix),
- $\mathbf{b}$ is an m-dimensional vector (the right-hand side),
- $\gtreqless$ is an m-dimensional vector of relations from $\{\leq,=,\geq\}$,
- $\mathbf{l}$ is an n-dimensional vector of lower bounds for $\mathbf{x}$, where $l_j\in\mathbb{R}\cup\{-\infty\}$ for all j
- $\mathbf{u}$ is an n-dimensional vector of upper bounds for $\mathbf{x}$, where $u_j\in\mathbb{R}\cup\{\infty\}$ for all j
- D is a symmetric positive-semidefinite $n\times n$ matrix (the quadratic objective function),
- $\mathbf{c}$ is an n-dimensional vector (the linear objective function), and
- c_0 is a constant.

If $D=0$, the program (QP) is actually a *linear program*. Section 10.2.2 on robustness briefly discusses the case of D not being positive-semidefinite and therefore not defining a convex program.

Solving the program means to find an n-vector $\mathbf{x}^*$ such that $A\mathbf{x}^* \gtreqless \mathbf{b}, \mathbf{l}\leq\mathbf{x}^*\leq\mathbf{u}$ (a *feasible solution*), and with the smallest objective function value ${\mathbf{x}^*}^T D\mathbf{x}^*+\mathbf{c}^T\mathbf{x}^*+c_0$ among all feasible solutions.

There might be no feasible solution at all, in which case the quadratic program is *infeasible*, or there might be feasible solutions of arbitrarily small objective function value, in which case the program is *unbounded*.

10.2 Design, Efficiency, and Robustness

The design of the package is quite simple. The linear or quadratic program to be solved is supplied in form of an object of a class that is a model of the concept *QuadraticProgram* (or some specialized other concepts, e.g. for linear programs). CGAL provides a number of easy-to-use and flexible models, see Section 10.3 below. The input data may be of any given number type, such as *double*, *int*, or any exact type.

Then the program is solved using the function *solve_quadratic_program* (or some specialized other functions, e.g. for linear programs). For this, you also have to provide a suitable *exact* number type *ET* used in the solution process. In case of input type *double*, solution methods that use floating-point-filtering are chosen by default for certain programs (in some cases, this is not appropriate, and the default should be changed; see Section 10.8 for details).

The output of this is an object of *Quadratic_program_solution<ET>* which you can in turn query for various things: what is the status of the program (optimally solved, infeasible, or unbounded?), what are the values of the optimal solution $\mathbf{x}^*$, what is the associated objective function value, etc.

You can in particular get *certificates* for the solution. In short, these are proofs that the output is correct. Thus, if you don't believe in the solution (whether it says "optimally solved", "infeasible", or "unbounded"), you can verify it yourself by using the certificates. Section 10.7 says more about this.

10.2.1 Efficiency

The concept *QuadraticProgram* (as well as the other specialized ones) require a *dense interface* of the program, in terms of *random-access iterators* over the matrices and vectors of (QP). Zero entries therefore play no special role and are treated like all other entries by the interface.

This has mainly historical reasons: the original motivation behind this package was low-dimensional geometric optimization where a dense representation is appropriate and efficient. In fact, the CGAL packages *Min_annulus_d<Traits>* and *Polytope_distance_d<Traits>* internally use the linear and quadratic programming solver.

As a user, however, you don't necessarily have to provide a dense *representation* of your program. You do not pass vectors or matrices to the solution functions, but rather specify the vectors and matrices through iterators. The iterator abstraction easily allows to build models that convert a sparse representation into a dense interface. The predefined models *Quadratic_program<NT>* and *Quadratic_program_from_mps<NT>* do exactly this; in using them, you can forget about the dense interface.

Nevertheless, if you care about efficiency, you cannot completely ignore the issue. If you think about a quadratic program in n variables and m constraints, its dense interface has $\Theta(n^2+mn)$ entries, even if actually very few of them are nonzero. This has consequences for the complexity of the internal computations. In fact, a single iteration of the solution process has complexity at least $\Omega(mn)$, since usually, all entries of the matrix A are accessed. This implies that problems where $\min(n,m)$ is large cannot be solved efficiently, even if the number of nonzero entries in the problem description is very small.

We can actually be quite precise about performance, in terms of the following parameters.

n	:	the number of variables (or columns of A),
m	:	the number of constraints (or rows of A),
e	:	the number of equality constraints,
r	:	the rank of the quadratic objective function matrix D.

The time required to solve the problems is in most cases linear in $\max(n,m)$, but with a factor heavily depending on $\min(n,e)+r$. Therefore, the solver will be efficient only if $\min(n,e)+r$ is small.

Here are the scenarios in which this applies:

- Quadratic programs with a small number of variables, but possibly a large number of inequality constraints,

- Linear programs with a small number of equality constraints but possibly a large number of variables,

- Quadratic programs with a small number of equality constraints and D of small rank, but possibly with a large number of variables.

How small is small? If $\min(n,e)+r$ is up to 10, the solver will probably be very fast, even if $\max(n,m)$ goes into the millions. If $\min(n,m)+r$ is up to a few hundreds, you may still get a solution within reasonable time, depending on the problem characteristics.

If you have a problem where both n and e are well above $1,000$, say, then chances are high that CGAL cannot solve it within reasonable time.

10.2.2 Robustness

Given that you use an *exact number type* in the function *solve_quadratic_program* (or in the other, specialized solution functions), the solver will give you *exact rational output*, for *every* convex quadratic program. It may fail to compute a solution only if

1. The quadratic program is too large (see the previous subsection on efficiency).
2. The quadratic objective function matrix D is not positive-semidefinite (see the discussion below).
3. The floating-point filter used by default for certain programs and input type *double* fails due to a *double* exponent overflow. This happens in rare cases only, and it does not pay off to sacrifice the efficiency of the filtered approach in order to cope with these rare cases. There are means, however, to avoid such problems by switching to a slower non-filtered variant, see Section 10.8.1.
4. The solver internally cycles. This also happens in rare cases only. However, if you have a hunch that the solver cycles on your problem, there are means to switch to a slower variant that is guaranteed not to cycle, see Section 10.8.2.

The second item merits special attention. First, you may ask why the solver does not check that D is positive semidefinite. But recall that D is given by a dense interface, and it would therefore cost $\Omega(n^2)$ time already to access all entries of the matrix D. The solver itself gets away with accessing much less entries of D in the relevant case where r, the rank of D, is small.

Nevertheless, the solver contains some runtime checks that may detect that the matrix D is not positive-semidefinite. But you may as well get an "optimal solution" in this case, even with valid certificates. The validity of these certificates, however, depends on D being positive-semidefinite; if this is not the case, the certificates only prove that the solver has found a "critical point" of your (nonconvex) program, but there are no guarantees whatsoever that this is a global optimum, or even a local optimum.

10.3 How to Enter and Solve a Program

In this section, we describe how you can supply and solve your problem, using the CGAL program models and solution functions. There are two essentially different ways to proceed, and we will discuss them in turn. In short,

- you can let the model take care of your program data; you start from an empty program and then simply insert the non-zero entries, or read them from a file (more generally, any input stream) in *MPSFormat*. You can also change program entries at any time. This is usually the most convenient way if you don't want to care about representation issues;
- you can maintain the data yourself and only supply suitable random-access iterators over the matrices and vectors. This is advantageous if you already have the data (explicitly, or implicitly encoded, for example through iterators) and want to avoid copying of data. Typically, this happens if you write generic iterator-based code.

Our running example is the following quadratic program in two variables:

$$\begin{array}{llcl} \text{minimize} & x^2+4(y-4)^2 & (= & x^2+4y^2-32y+64) \\ \text{subject to} & x+y & \leq & 7 \\ & -x+2y & \leq & 4 \\ & x & \geq & 0 \\ & y & \geq & 0 \\ & y & \leq & 4 \end{array}$$

Figure 10.1 shows a picture. It depicts the five inequalities of the program, along with the *feasible region* (green), the set of points that satisfy all the five constraints. The dashed elliptic curves represent *contour lines* of the objective function, i.e., along each dashed curve, the objective function value is constant.

The global minimum of the objective function is attained at the point $(0,4)$, and the minimum within the feasible region appears at the point $(2,3)$ marked with a black dot. The value of the objective function at this optimal solution is $2^2+4(3-4)^2=8$.

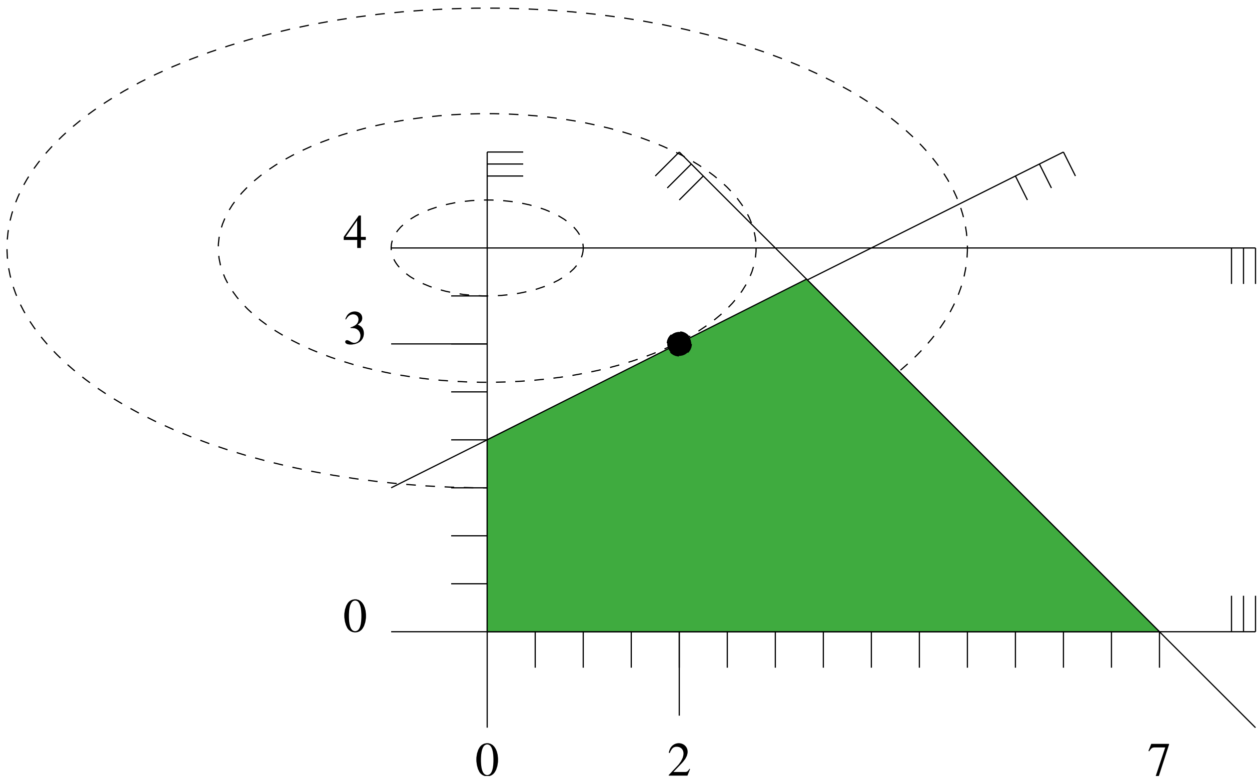


Figure 10.1: A quadratic program in two variables

10.3.1 Constructing a Program from Data

Here is how this quadratic program can be solved in CGAL according to the first way (letting the model take care of the data). We use *int* as the input type, and *MP_Float* or *Gmpz* (which is faster and preferable if GMP is installed) as the exact type for the internal computations. In larger examples, it pays off to use *double* as input type in order to profit from the automatic floating-point filtering that takes place then.

For examples how to work with the input type *double*, we refer to Sections 10.5 and 10.8.

Note: For the quadratic objective function, the entries of the matrix $2D$ have to be provided, rather than D. Although this is common to almost all quadratic programming solvers, it can easily be overlooked by a novice.

```
#include <iostream>
#include <cassert>
#include <CGAL/basic.h>
#include <CGAL/QP_models.h>
#include <CGAL/QP_functions.h>

// choose exact integral type
#ifdef CGAL_USE_GMP
#include <CGAL/Gmpz.h>
typedef CGAL::Gmpz ET;
#else
#include <CGAL/MP_Float.h>
typedef CGAL::MP_Float ET;
#endif

// program and solution types
typedef CGAL::Quadratic_program<int> Program;
typedef CGAL::Quadratic_program_solution<ET> Solution;

int main() {
  // by default, we have a nonnegative QP with Ax <= b
  Program qp (CGAL::SMALLER, true, 0, false, 0);

  // now set the non-default entries:
  const int X = 0;
  const int Y = 1;
  qp.set_a(X, 0,  1); qp.set_a(Y, 0, 1); qp.set_b(0, 7);  //  x + y  <= 7
  qp.set_a(X, 1, -1); qp.set_a(Y, 1, 2); qp.set_b(1, 4);  // -x + 2y <= 4
  qp.set_u(Y, true, 4);                                   //       y <= 4
  qp.set_d(X, X, 2); qp.set_d (Y, Y, 8); // !!specify 2D!!    x^2 + 4 y^2
  qp.set_c(Y, -32);                                       // -32y
  qp.set_c0(64);                                          // +64

  // solve the program, using ET as the exact type
  Solution s = CGAL::solve_quadratic_program(qp, ET());
  assert (s.solves_quadratic_program(qp));

  // output solution
  std::cout << s;
  return 0;
}
```

File: examples/QP_solver/first_qp.cpp

Asuming that GMP is installed, the output of the of the above program is:

```
status:          OPTIMAL
objective value: 8/1
variable values:
```

```
0: 2/1
1: 3/1
```

If GMP is not installed, the values are of course the same, but numerator and denominator might have a common divisor that is not factored out.

10.3.2 Constructing a Program from a Stream

Here, the program data must be available in *MPSFormat* (the *MPSFormat* page shows how our running example looks like in this format, and it briefly explains the format). Assuming that your working directory contains the file first_qp.mps, the following program will read and solve it, with the same output as before.

```cpp
#include <iostream>
#include <fstream>
#include <CGAL/basic.h>
#include <CGAL/QP_models.h>
#include <CGAL/QP_functions.h>

// choose exact integral type
#ifdef CGAL_USE_GMP
#include <CGAL/Gmpz.h>
typedef CGAL::Gmpz ET;
#else
#include <CGAL/MP_Float.h>
typedef CGAL::MP_Float ET;
#endif

// program and solution types
typedef CGAL::Quadratic_program_from_mps<int> Program;
typedef CGAL::Quadratic_program_solution<ET> Solution;

int main() {
  std::ifstream in ("first_qp.mps");
  Program qp(in);          // read program from file
  assert (qp.is_valid()); // we should have a valid mps file

  // solve the program, using ET as the exact type
  Solution s = CGAL::solve_quadratic_program(qp, ET());

  // output solution
  std::cout << s;
  return 0;
}
```

File: examples/QP_solver/first_qp_from_mps.cpp

10.3.3 Constructing a Program from Iterators

The following program again solves our running example from above, with the same output, but this time with iterators over data stored in suitable containers. You can see that we also store zero entries here (in D). For this toy problem, the previous two approaches (program from data/stream) are clearly preferable, but Section 10.5 shows an example where it makes sense to use the iterator-based approach.

```
#include <iostream>
#include <CGAL/basic.h>
#include <CGAL/QP_models.h>
#include <CGAL/QP_functions.h>

// choose exact integral type
#ifdef CGAL_USE_GMP
#include <CGAL/Gmpz.h>
typedef CGAL::Gmpz ET;
#else
#include <CGAL/MP_Float.h>
typedef CGAL::MP_Float ET;
#endif

// program and solution types
typedef CGAL::Quadratic_program_from_iterators
<int**,                                                // for A
 int*,                                                 // for b
 CGAL::Const_oneset_iterator<CGAL::Comparison_result>, // for r
 bool*,                                                // for fl
 int*,                                                 // for l
 bool*,                                                // for fu
 int*,                                                 // for u
 int**,                                                // for D
 int*>                                                 // for c
Program;
typedef CGAL::Quadratic_program_solution<ET> Solution;

int main() {
  int   Ax[] = {1, -1};                        // column for x
  int   Ay[] = {1,  2};                        // column for y
  int*  A[] = {Ax, Ay};                        // A comes columnwise
  int   b[] = {7, 4};                          // right-hand side
  CGAL::Const_oneset_iterator<CGAL::Comparison_result>
        r(    CGAL::SMALLER);                  // constraints are "<="
  bool fl[] = {true, true};                    // both x, y are
   lower-bounded
  int   l[] = {0, 0};
  bool fu[] = {false, true};                   // only y is upper-bounded
  int   u[] = {0, 4};                          // x's u-entry is ignored
  int  D1[] = {2};                             // 2D_{1,1}
  int  D2[] = {0, 8};                          // 2D_{2,1}, 2D_{2,2}
  int*  D[] = {D1, D2};                        // D-entries on/below
   diagonal
  int   c[] = {0, -32};
  int   c0  = 64;                              // constant term

  // now construct the quadratic program; the first two parameters are
  // the number of variables and the number of constraints (rows of A)
  Program qp (2, 2, A, b, r, fl, l, fu, u, D, c, c0);

  // solve the program, using ET as the exact type
```

```
  Solution s = CGAL::solve_quadratic_program(qp, ET());

  // output solution
  std::cout << s;
  return 0;
}
```

File: `examples/QP_solver/first_qp_from_iterators.cpp`

Note 1: The example shows an interesting feature of this approach: not all data need to come from containers. Here, the iterator over the vector of relations can be provided through the class *Const_oneset_iterator<T>*, since all entries of this vector are equal to *CGAL::SMALLER*. The same could have been done with the vector *fl* for the finiteness of the lower bounds.

Note 2: The program type looks a bit scary, with its total of 9 template arguments, one for each iterator type. In Section 10.5.1 we show how the explicit construction of this type can be circumvented.

10.4 Solving Linear and Nonnegative Programs

Let us reconsider the general form of (QP) from Section 10.1 above. If $D = 0$, the quadratic program is in fact a *linear program*, and in the case that the bound vectors l is the zero vector and all entries of u are ∞, the program is said to be *nonnegative*. The package offers dedicated models and solution methods for these special cases.

From an interface perspective, this is just syntactic sugar: in the model *Quadratic_program<NT>*, we can easily set the default bounds so that a nonnegative program results, and a linear program is obtained by simply not inserting any D-entries. Even in the iterator-based approach (see *QP_solver/first_qp_from_iterators.cpp*), linear and nonnegative programs can easily be defined through suitable *Const_oneset_iterator<T>*-style iterators.

The main reason for having dedicated solution methods for linear and nonnegative programs is efficiency: if the solver knows that the program is linear, it can save some computations compared to the general solver that unknowingly has to fiddle around with a zero D-matrix. As in Section 10.2.2 above, we can argue that checking in advance whether $D = 0$ is not an option in general, since this may require $\Omega(n^2)$ time on the dense interface.

Similarly, if the solver knows that the program is nonnegative, it will be more efficient than under the general bounds $\mathbf{l} \leq \mathbf{x} \leq \mathbf{u}$. You can argue that nonnegativity *is* something that could easily be checked in time $O(n)$ beforehand, but then again nonnegative programs are so frequent that the syntactic sugar aspect becomes somewhat important. After all, we can save four iterators in specifying a nonnegative linear program in terms of the concept *NonnegativeLinearProgram* rather than *LinearProgram*.

Often, there are no bounds at all for the variables, i.e., all entries of $\mathbf{l}$ are $-\infty$, and all entries of $\mathbf{u}$ are ∞ (this is called a *free* program). There is no dedicated solution method for this case (a free quadratic or linear program is treated like a general quadratic or linear program), but all predefined models make it easy to specify all sorts of default bounds, covering the free case.

10.4.1 The Linear Programming Solver

Let's go back to our first quadratic program from above and change it into a linear program by simply removing the quadratic part of the objective function:

$$\begin{array}{lrcl}
\text{minimize} & -32y+64 & & \\
\text{subject to} & x+y & \leq & 7 \\
& -x+2y & \leq & 4 \\
& x & \geq & 0 \\
& y & \geq & 0 \\
& y & \leq & 4
\end{array}$$

Figure 10.2 shows how this looks like. We will not visualize a linear objective function with contour lines but with arrows instead. The arrow represents the (direction) of the vector $-c$, and we are looking for a feasible solution that is "extreme" in the direction of the arrow. In our small example, this is the unique point "on" the two constraints $x_1+x_2 \leq 7$ and $-x_1+x_2 \leq 4$, the point $(10/3, 11/3)$ marked with a black dot. The optimal objective function value is $-32(11/3)+64 = -160/3$.

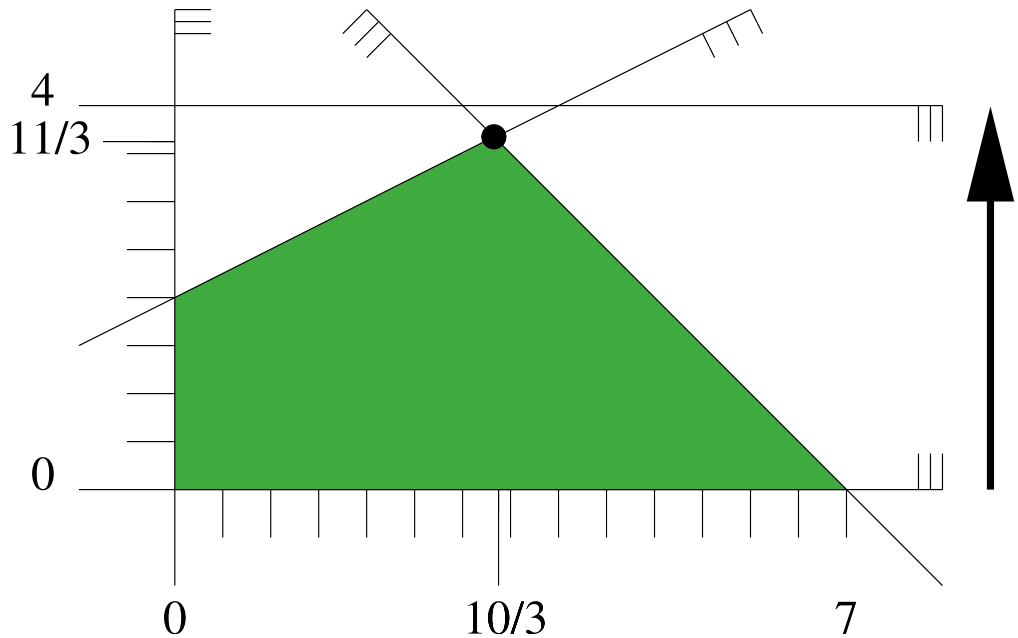


Figure 10.2: A linear program in two variables

Here is CGAL code for solving it, using the dedicated LP solver, and according to the three ways for constructing a program that we have already discussed in Section 10.3.

QP_solver/first_lp.cpp
QP_solver/first_lp_from_mps.cpp
QP_solver/first_lp_from_iterators.cpp

In all cases, the output is

```
status:          OPTIMAL
objective value: -160/3
variable values:
  0: 10/3
  1: 11/3
```

10.4.2 The Nonnegative Quadratic Programming Solver

If we go back to our first quadratic program and remove the constraint $y \leq 4$, we arrive at a nonnegative quadratic program:

$$\begin{array}{lrcl} \text{minimize} & x^2+4(y-4)^2 & (= & x^2+4y^2-32y+64) \\ \text{subject to} & x+y & \leq & 7 \\ & -x+2y & \leq & 4 \\ & x,y & \geq & 0 \end{array}$$

Figure 10.3 contains the illustration; since the constraint $y \leq 4$ was redundant, the feasible region and the optimal solution do not change.

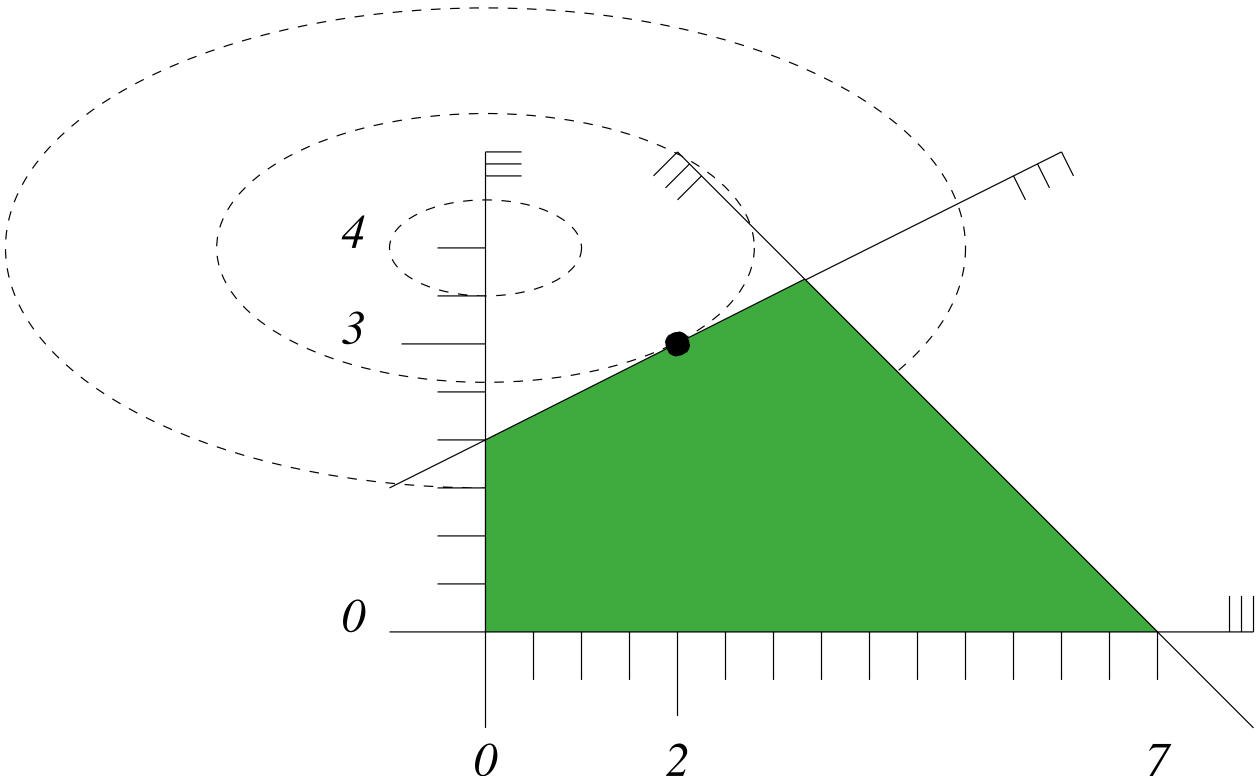


Figure 10.3: A nonnegative quadratic program in two variables

The following programs (using the dedicated solver for nonnegative quadratic programs) will therefore again output

```
status:          OPTIMAL
objective value: 8/1
variable values:
  0: 2/1
  1: 3/1
```

QP_solver/first_nonnegative_qp.cpp
QP_solver/first_nonnegative_qp_from_mps.cpp
QP_solver/first_nonnegative_qp_from_iterators.cpp

10.4.3 The Nonnegative Linear Programming Solver

Finally, a dedicated model and function is available for nonnnegative linear programs as well. Let's take our linear program from above and remove the constraint $y \leq 4$ to obtain a nonnegative linear program. At the same time we remove the constant objective function term to get a "minimal" input and a "shortest" program; the optimal value is $-32(11/3) = -352/3$.

$$\begin{array}{lrcl} \text{minimize} & -32y & & \\ \text{subject to} & x+y & \leq & 7 \\ & -x+2y & \leq & 4 \\ & x,y & \geq & 0 \end{array}$$

This can be solved by any of the following three programs

QP_solver/first_nonnegative_lp.cpp
QP_solver/first_nonnegative_lp_from_mps.cpp
QP_solver/first_nonnegative_lp_from_iterators.cpp

The output will always be

```
status:          OPTIMAL
objective value: -352/3
variable values:
  0: 10/3
  1: 11/3
```

10.5 Working from Iterators

Here we present a somewhat more advanced example that emphasizes the usefulness of solving linear and quadratic programs from iterators. Let's look at a situation in which a linear program is given implicitly, and access to it is gained through properly constructed iterators.

The problem we are going to solve is the following: given points $p_1, \ldots p_n$ in d-dimensional space and another point p: is p in the convex hull of $\{p_1, \ldots, p_n\}$? In formulas, this is the case if and only if there are real coefficients $\lambda_1, \ldots, \lambda_n$ such that p is a convex combination of $p_1, \ldots, p_n$:

$$p = \sum_{j=1}^{n} \lambda_j \, p_j, \quad \sum_{j=1}^{n} \lambda_j = 1, \quad \lambda_j \geq 0 \text{ for all } j.$$

The problem of testing the existence of such λ_j can be expressed as a linear program. It becomes particularly easy when we use the homogeneous representations of the points: if $q_1, \ldots, q_n, q \in \mathbb{R}^{d+1}$ are homogeneous coordinates for $p_1, \ldots, p_n, p$ with positive homogenizing coordinates $h_1, \ldots, h_n, h$, we have

$$q_j = h_j \cdot (p_j \mid 1) \text{ for all } j, \text{ and } q = h \cdot (p \mid 1).$$

Now, nonnegative $\lambda_1, \ldots, \lambda_n$ are suitable coefficients for a convex combination if and only if

$$\sum_{j=1}^{n} \lambda_j (p_j \mid 1) = (p \mid 1),$$

equivalently, if there are $\mu_1, \ldots, \mu_n$ (with $\mu_j = \lambda_j \cdot h/h_j$ for all j) such that

$$\sum_{j=1}^{n} \mu_j\, q_j = q, \quad \mu_j \geq 0 \text{ for all } j.$$

The linear program now tests for the existence of nonnegative μ_j that satisfy the latter equation. Below is the code; it defines a function that solves the linear program, given p and $p_1, \ldots, p_n$ (through an iterator range). The only (mild) trickery involved is the construction of the nested iterator through a fixed column of the constraint matrix A. We get this from transforming the iterator through the points using a functor that maps a point to an iterator through its homogeneous coordinates.

```
#include <boost/config.hpp>
#include <boost/iterator/transform_iterator.hpp>
#include <CGAL/Kernel_traits.h>
#include <CGAL/QP_models.h>
#include <CGAL/QP_functions.h>

// unary function to get homogeneous begin-iterator of point
template <class Point_d>
struct Homogeneous_begin  {
  typedef typename Point_d::Homogeneous_const_iterator result_type;
  result_type operator() (const Point_d& p) const {
    return p.homogeneous_begin();
  }
};

// function to solve the LP that tests whether a point is in the
// convex hull of other points; the type ET is an exact type used
// for the internal computations
template <class Point_d, class RandomAccessIterator, class ET>
CGAL::Quadratic_program_solution<ET>
solve_convex_hull_containment_lp (const Point_d& p,
                                  RandomAccessIterator begin,
                                  RandomAccessIterator end, const ET&
    dummy)
{
  // Constraint matrix type: A[j][i] is the i-th homogeneous coordinate
    of p_j
  typedef boost::transform_iterator
    <Homogeneous_begin<Point_d>, RandomAccessIterator> A_it;
  // Right-hand side type: b[i] is the i-th homogeneous coordinate of p
  typedef typename Point_d::Homogeneous_const_iterator B_it;
  // Relation type ("=")
  typedef CGAL::Const_oneset_iterator<CGAL::Comparison_result> R_it;
  // input number type
  typedef typename CGAL::Kernel_traits<Point_d>::Kernel::RT RT;
  // Linear objective function type (c=0: we only test feasibility)
  typedef CGAL::Const_oneset_iterator<RT> C_it;
  // the nonnegative linear program type
  typedef
    CGAL::Nonnegative_linear_program_from_iterators<A_it, B_it, R_it,
   C_it>
    Program;
```

```
  // ok, we are prepared now: construct program and solve it
  Program lp (end-begin,          // number of variables
              p.dimension()+1,    // number of constraints
              A_it (begin), B_it (p.homogeneous_begin()),
              R_it (CGAL::EQUAL), C_it (0));
  return CGAL::solve_nonnegative_linear_program (lp, dummy);
}
```

File: examples/QP_solver/solve_convex_hull_containment_lp.h

To see this in action, let us call it with $p_1 = (0,0), p_2 = (10,0), p_3 = (0,10)$ fixed (they define a triangle) and all integral points p in $[0,10]^2$. We know that p is in the convex hull of $\{p_1, p_2, p_3\}$ if and only if its two coordinates sum up to 10 at most. As the exact type, we use *MP_Float* or *Gmpzf* (which is faster and preferable if GMP is installed).

```
#include <cassert>
#include <vector>
#include <CGAL/Cartesian_d.h>
#include <CGAL/MP_Float.h>
#include "solve_convex_hull_containment_lp.h"

// choose exact floating-point type
#ifdef CGAL_USE_GMP
#include <CGAL/Gmpzf.h>
typedef CGAL::Gmpzf ET;
#else
#include <CGAL/MP_Float.h>
typedef CGAL::MP_Float ET;
#endif

typedef CGAL::Cartesian_d<double> Kernel_d;
typedef Kernel_d::Point_d Point_d;

bool is_in_convex_hull (const Point_d& p,
                        std::vector<Point_d>::const_iterator begin,
                        std::vector<Point_d>::const_iterator end)
{
  CGAL::Quadratic_program_solution<ET> s =
    solve_convex_hull_containment_lp (p, begin, end, ET(0));
  return !s.is_infeasible();
}

int main()
{
  std::vector<Point_d> points;
  // convex hull: simplex spanned by {(0,0), (10,0), (0,10)}
  points.push_back (Point_d ( 0.0,  0.0));
  points.push_back (Point_d (10.0,  0.0));
  points.push_back (Point_d ( 0.0, 10.0));
  for (int i=0; i<=10; ++i)
    for (int j=0; j<=10; ++j) {
      // (i,j) is in the simplex iff i+j <= 10
      bool contained = is_in_convex_hull
```

```
        (Point_d (i, j), points.begin(), points.end());
      assert (contained == (i+j<=10));
    }

  return 0;
}
```

File: examples/QP_solver/convex_hull_containment.cpp

10.5.1 Using Makers

You already noticed in the previous example that the actual template arguments for *CGAL::Nonnegative_linear_program_from_iterators<A_it, B_it, R_it, C_it>* can be quite elaborate, and this only gets worse if you plug more iterators into each other. In general, you want to construct a program from given expressions for the iterators, but the types of these expressions are probably very complicated and difficult to look up.

You can avoid the explicit construction of the type *CGAL::Nonnegative_linear_program_from_iterators<A_it, B_it, R_it, C_it>* if you only need an expression of it, e.g. to pass it directly as an argument to the solving function. Here is an alternative version of *QP_solver/solve_convex_hull_containment_lp.h* that shows how this works. In effect, you get shorter and more readable code.

```
#include <boost/config.hpp>
#include <boost/iterator/transform_iterator.hpp>
#include <CGAL/Kernel_traits.h>
#include <CGAL/QP_models.h>
#include <CGAL/QP_functions.h>

// unary function to get homogeneous begin-iterator of point
template <class Point_d>
struct Homogeneous_begin  {
  typedef typename Point_d::Homogeneous_const_iterator result_type;
  result_type operator() (const Point_d& p) const {
    return p.homogeneous_begin();
  }
};

// function to test whether point is in the convex hull of other points;
// the type ET is an exact type used for the computations
template <class Point_d, class RandomAccessIterator, class ET>
CGAL::Quadratic_program_solution<ET>
solve_convex_hull_containment_lp (const Point_d& p,
                                  RandomAccessIterator begin,
                                  RandomAccessIterator end, const ET&
   dummy)
{
  // construct program and solve it
  return CGAL::solve_nonnegative_linear_program
    (CGAL::make_nonnegative_linear_program_from_iterators
     (end-begin,
     // n
      p.dimension()+1,
     // m
```

```
      boost::transform_iterator
      <Homogeneous_begin<Point_d>, RandomAccessIterator>(begin),
     // A
      typename Point_d::Homogeneous_const_iterator
    (p.homogeneous_begin()),// b
      CGAL::Const_oneset_iterator<CGAL::Comparison_result>(CGAL::EQUAL),
     // ~
      CGAL::Const_oneset_iterator
      <typename CGAL::Kernel_traits<Point_d>::Kernel::RT> (0)), dummy);
     // c
}
```

File: examples/QP_solver/solve_convex_hull_containment_lp2.h

10.6 Important Variables and Constraints

If you have a solution $\mathbf{x}^*$ of a linear or quadratic program, the "important" variables are typically the ones that are not on their bounds. In case of a nonnegative program, these are the nonzero variables. Going back to the example of the previous Section 10.5, we can easily interpret their importance: the nonzero variables correspond to points p_j that actually contribute to the convex combination that yields p.

The following example shows how we can access the important variables, using the iterators *basic_variable_indices_begin()* and *basic_variable_indices_end()*.

We generate a set of points that form a 4-gon in $[0,4]^2$, and then find the ones that contribute to the convex combinations of all 25 lattice points in $[0,4]^2$. If the lattice point in question is not in the 4-gon, we simply output this fact.

```
#include <cassert>
#include <vector>
#include <CGAL/Cartesian_d.h>
#include <CGAL/MP_Float.h>
#include "solve_convex_hull_containment_lp2.h"

typedef CGAL::Cartesian_d<double> Kernel_d;
typedef Kernel_d::Point_d Point_d;
typedef CGAL::Quadratic_program_solution<CGAL::MP_Float> Solution;

int main()
{
  std::vector<Point_d> points;
  // convex hull: 4-gon spanned by {(1,0), (4,1), (4,4), (2,3)}
  points.push_back (Point_d (1, 0)); // point 0
  points.push_back (Point_d (4, 1)); // point 1
  points.push_back (Point_d (4, 4)); // point 2
  points.push_back (Point_d (2, 3)); // point 3

  // test all 25 integer points in [0,4]^2
  for (int i=0; i<=4; ++i)
    for (int j=0; j<=4; ++j) {
      Point_d p (i, j);
```

```
      Solution s = solve_convex_hull_containment_lp
        (p, points.begin(), points.end(), CGAL::MP_Float());
      std::cout << p;
      if (s.is_infeasible())
        std::cout << " is not in the convex hull\n";
      else {
        assert (s.is_optimal());
        std::cout << " is a convex combination of the points ";
        Solution::Index_iterator it = s.basic_variable_indices_begin();
        Solution::Index_iterator end = s.basic_variable_indices_end();
        for (; it != end; ++it) std::cout << *it << " ";
        std::cout << std::endl;
      }
    }
  return 0;
}
```

File: `examples/QP_solver/important_variables.cpp`

It turns out that exactly three of the four points contribute to any convex combination, even through there are lattice points that lie in the convex hull of less than three of the points. This shows that the set of basic variables that we access in the example does not necessarily coincide with the set of important variables as defined above. In fact, it is only guaranteed that a non-basic variable attains one of its bounds, but there might be basic variables that also have this property. In linear and quadratic programming terms, such a situation is called a *degeneracy*.

There is also the concept of an important constraint: this is typically a constraint in the system $A\mathbf{x} \gtreqless \mathbf{b}$ that is satisfied with equality at $\mathbf{x}^*$. Program *QP_solver/first_qp_basic_constraints.cpp* shows how these can be accessed, using the iterators *basic_constraint_indices_begin()* and *basic_constraint_indices_end()*.

Again, we have a disagreement between "basic" and "important": it is guaranteed that all basic constraints are satisfied with equality at $\mathbf{x}^*$, but there might be non-basic constraints that are satisfied with equality as well.

10.7 Solution Certificates

Suppose the solver tells you that the problem you have entered is infeasible. Why should you believe this? Similarly, you can quite easily verify that a claimed optimal solution is feasible, but why is there no better one?

Certificates are proofs that the solver can give you in order to convince you that what it claims is indeed true. The archetype of such a proof is *Farkas Lemma* [MG06].

Farkas Lemma: *Either* the inequality system

$$\begin{array}{rcl} A\mathbf{x} & \leq & \mathbf{b} \\ \mathbf{x} & \geq & 0 \end{array}$$

has a solution $\mathbf{x}^*$, *or* there exists a vector $\mathbf{y}$ such that

$$\begin{array}{rcl} \mathbf{y} & \geq & 0 \\ \mathbf{y}^T A & \geq & 0 \\ \mathbf{y}^T \mathbf{b} & < & 0, \end{array}$$

but not both.

Thus, if someone wants to convince you that the first system in the Farkas Lemma is infeasible, that person can simply give you a vector **y** that solves the second system. Since you can easily verify yourself that the **y** you got satisfies this second system, you now have a certificate for the infeasibility of the first system, assuming that you believe in Farkas Lemma.

Here we show how the solver can convince you. We first set up an infeasible linear program with constraints of the type $A\mathbf{x} \leq \mathbf{b}, \mathbf{x} \geq 0$; then we solve it and ask for a certificate. Finally, we verify the certificate by simply checking the inequalities of the second system in Farkas Lemma.

```
#include <cassert>
#include <CGAL/basic.h>
#include <CGAL/QP_models.h>
#include <CGAL/QP_functions.h>

// choose exact integral type
#ifdef CGAL_USE_GMP
#include <CGAL/Gmpz.h>
typedef CGAL::Gmpz ET;
#else
#include <CGAL/MP_Float.h>
typedef CGAL::MP_Float ET;
#endif

// program and solution types
typedef CGAL::Nonnegative_linear_program_from_iterators
<int**,                                                   // for A
 int*,                                                    // for b
 CGAL::Comparison_result*,                                // for r
 int*>                                                    // for c
Program;
typedef CGAL::Quadratic_program_solution<ET> Solution;

// we demonstrate Farkas Lemma: either the system
//     A x <= b
//       x >= 0
// has a solution, or there exists y such that
//       y >= 0
//    y^TA >= 0
//    y^Tb <  0
// In the following instance, the first system has no solution,
// since adding up the two inequalities gives x_2 <= -1:
//     x_1 - 2x_2  <=  1
//    -x_1 + 3x_2  <= -2
//     x_1,   x_2  >=  0

int main() {
  int  Ax1[] = { 1, -1};                           // column for x1
  int  Ax2[] = {-2,  3};                           // column for x2
  int*   A[] = {Ax1, Ax2};                         // A comes columnwise
  int    b[] = {1, -2};                            // right-hand side
  CGAL::Comparison_result
    r[] = {CGAL::SMALLER, CGAL::SMALLER};         // constraints are "<="
  int    c[] = {0, 0};                            // zero objective function

  // now construct the linear program; the first two parameters are
```

```
  // the number of variables and the number of constraints (rows of A)
  Program lp (2, 2, A, b, r, c);

  // solve the program, using ET as the exact type
  Solution s = CGAL::solve_nonnegative_linear_program(lp, ET());

  // get certificate for infeasibility
  assert (s.is_infeasible());
  Solution::Infeasibility_certificate_iterator y =
    s.infeasibility_certificate_begin();
  // check y >= 0
  assert (y[0] >= 0);
  assert (y[1] >= 0);
  // check y^T A >= 0
  assert (y[0] * A[0][0] + y[1] * A[0][1] >= 0);
  assert (y[0] * A[1][0] + y[1] * A[1][1] >= 0);
  // check y^T b < 0
  assert (y[0] * b[0] + y[1] * b[1] < 0);

  return 0;
}
```

File: examples/QP_solver/infeasibility_certificate.cpp

There are similar certificates for optimality and unboundedness that you can see in action in the programs *QP_solver/optimality_certificate.cpp* and *QP_solver/unboundedness_certificate.cpp*. The underlying variants of Farkas Lemma are somewhat more complicated, due to the mixed relations in $\gtreqless$ and the general bounds. The certificate section of *Quadratic_program_solution<ET>* gives the full picture and mathematically proves the correctness of the certificates.

10.8 Customizing the Solver

Sometimes it is necessary to alter the default behavior of the solver. This can be done by passing a suitably prepared object of the class *Quadratic_program_options* to the solution functions. Most options concern "soft" issues like verbosity, but there are two notable case where it is of critical importance to be able to change the defaults.

10.8.1 Exponent Overflow in Double Using Floating-Point Filters

The filtered version of the solver that is used for some problems by default on input type *double* internally constructs double-approximations of exact multiprecision values. If these exact values are extremely large, this may lead to *infinite double* values and incorrect results. In debug mode, the solver will notice this through a certificate cross-check in the end (or even earlier). In this case, it is advisable to explicitly switch to a non-filtered *pricing strategy*, see *Quadratic_program_pricing_strategy*.

Hint: If you have a program where the number of variables n and the number of constraints m have the same order of magnitude, the filtering will usually have no dramatic effect on the performance, so in that case you might as well switch to *QP_PARTIAL_DANTZIG* to be safe from the issue described here (see *QP_solver/cycling.cpp* for an example that shows how to change the pricing strategy).

10.8.2 The Solver Internally Cycles

Consider the following program. It reads a nonnegative linear program from the file `cycling.mps` (which is in the example directory as well), and then solves it in verbose mode, using *Bland's rule*, see *Quadratic_program_pricing_strategy*.

```
#include <iostream>
#include <fstream>
#include <CGAL/basic.h>
#include <CGAL/QP_models.h>
#include <CGAL/QP_functions.h>

// choose exact floating-point type
#ifdef CGAL_USE_GMP
#include <CGAL/Gmpzf.h>
typedef CGAL::Gmpzf ET;
#else
#include <CGAL/MP_Float.h>
typedef CGAL::MP_Float ET;
#endif

// program and solution types
typedef CGAL::Quadratic_program_from_mps<double> Program;
typedef CGAL::Quadratic_program_solution<ET> Solution;

int main() {
  std::ifstream in ("cycling.mps");
  Program lp(in);          // read program from file
  assert (lp.is_valid()); // we should have a valid mps file...
  assert (lp.is_linear());      // ... and it should be linear...
  assert (lp.is_nonnegative()); // as well as nonnegative

  // solve the program, using ET as the exact type
  // choose verbose mode and Bland pricing
  CGAL::Quadratic_program_options options;
  options.set_verbosity(1);                         // verbose mode
  options.set_pricing_strategy(CGAL::QP_BLAND);     // Bland's rule
  options.set_auto_validation(true);                // automatic
    self-check
  Solution s = CGAL::solve_nonnegative_linear_program(lp, ET(), options);
  assert (s.is_valid());                      // did the self-check
    succeed?

  // output solution
  std::cout << s;
  return 0;
}
```

File: examples/QP_solver/cycling.cpp

If you comment the line

```
options.set_pricing_strategy(CGAL::QP_BLAND);          // Bland's rule
```

you will see that the solver cycles: the verbose mode outputs the same sequence of six iterations over and over again. By switching to *QP_BLAND*, the solution process typically slows down a bit (it may also speed up in some cases), but now it is guaranteed that no cycling occurs.

In general, the verbose mode can be of use when you are not sure whether the solver "has died", or whether it simply takes very long to solve your problem. We refer to the class *Quadratic_program_options* for further details.

10.9 Some Benchmarks for Convex Hull Containment

Here we want to show what you can expect from the solver's performance in a specific application; we don't know whether this application is typical in your case, and we make no claims whatsoever about the performance in other applications.

Still, the example shows that the performance can be dramatically affected by switching between pricing strategies, and we give some hints on how to achieve good performance in general.

The application is the one already discussed in Section 10.5 above: testing whether a point is in the convex hull of other points. To be able to switch between pricing strategies, we add another parameter of type *Quadratic_program_options* to the function *solve_convex_hull_containment_lp* that we pass on to the solution function:

```
#include <boost/config.hpp>
#include <boost/iterator/transform_iterator.hpp>
#include <CGAL/Kernel_traits.h>
#include <CGAL/QP_options.h>
#include <CGAL/QP_models.h>
#include <CGAL/QP_functions.h>

// unary function to get homogeneous begin-iterator of point
template <class Point_d>
struct Homogeneous_begin  {
  typedef typename Point_d::Homogeneous_const_iterator result_type;
  result_type operator() (const Point_d& p) const {
    return p.homogeneous_begin();
  }
};

// function to test whether point is in the convex hull of other points;
// the type ET is an exact type used for the computations
template <class Point_d, class RandomAccessIterator, class ET>
CGAL::Quadratic_program_solution<ET>
solve_convex_hull_containment_lp (const Point_d& p,
                                  RandomAccessIterator begin,
                                  RandomAccessIterator end, const ET&
   dummy,
                                  const CGAL::Quadratic_program_options&
   o)
{
  // construct program and solve it
  return CGAL::solve_nonnegative_linear_program
    (CGAL::make_nonnegative_linear_program_from_iterators
     (end-begin,
      // n
```

```
      p.dimension()+1,
     // m
      boost::transform_iterator
      <Homogeneous_begin<Point_d>, RandomAccessIterator>(begin),
     // A
      typename Point_d::Homogeneous_const_iterator
    (p.homogeneous_begin()),// b
      CGAL::Const_oneset_iterator<CGAL::Comparison_result>(CGAL::EQUAL),
     // ~
      CGAL::Const_oneset_iterator
      <typename CGAL::Kernel_traits<Point_d>::Kernel::RT> (0)),
     // c
      dummy, o);
}
```

File: examples/QP_solver/solve_convex_hull_containment_lp3.h

Now let us test containment of the origin in the convex hull of n random points in $[0,1]^d$ (it will most likely not be contained, and it turns out that this is the most expensive case). In the program below, we use $d = 10$ and $n = 100,000$, and we comment on some other combinations of n and d below (feel free to experiment with still other values).

```
#include <vector>
#include <CGAL/Cartesian_d.h>
#include <CGAL/MP_Float.h>
#include <CGAL/Random.h>
#include <CGAL/Timer.h>
#include "solve_convex_hull_containment_lp3.h"

// choose exact floating-point type
#ifdef CGAL_USE_GMP
#include <CGAL/Gmpzf.h>
typedef CGAL::Gmpzf ET;
#else
#include <CGAL/MP_Float.h>
typedef CGAL::MP_Float ET;
#endif

typedef CGAL::Cartesian_d<double> Kernel_d;
typedef Kernel_d::Point_d Point_d;

int main()
{
  const int d = 10;       // change this in order to experiment
  const int n = 100000;   // change this in order to experiment

  // generate n random d-dimensional points in [0,1]^d
  CGAL::Random rd;
  std::vector<Point_d> points;
  for (int j =0; j<n; ++j) {
    std::vector<double> coords;
    for (int i=0; i<d; ++i)
      coords.push_back(rd.get_double());
```

```
    points.push_back (Point_d (d, coords.begin(), coords.end()));
  }

  // benchmark all pricing strategies in turn
  CGAL::Quadratic_program_pricing_strategy strategy[] = {
    CGAL::QP_CHOOSE_DEFAULT,              // QP_PARTIAL_FILTERED_DANTZIG
    CGAL::QP_DANTZIG,                     // Dantzig's pivot rule...
    CGAL::QP_PARTIAL_DANTZIG,             // ... with partial pricing
    CGAL::QP_BLAND,                       // Bland's pivot rule
    CGAL::QP_FILTERED_DANTZIG,            // Dantzig's filtered pivot
   rule...
    CGAL::QP_PARTIAL_FILTERED_DANTZIG     // ... with partial pricing
  };

  CGAL::Timer t;
  for (int i=0; i<6; ++i) {
    // test strategy i
    CGAL::Quadratic_program_options options;
    options.set_pricing_strategy (strategy[i]);
    t.reset(); t.start();
    // is origin in convex hull of the points? (most likely, not)
    solve_convex_hull_containment_lp
      (Point_d (d, CGAL::ORIGIN), points.begin(), points.end(),
       ET(0), options);
    t.stop();
    std::cout << "Time (s) = " << t.time() << std::endl;
  }

  return 0;
}
```

File: examples/QP_solver/convex_hull_containment_benchmarks.cpp

If you compile with the macros NDEBUG or CGAL_QP_NO_ASSERTIONS set (this is essential for good performance!!), you will see runtimes that qualitatively look as follows (on your machine, the actual runtimes will roughly be some fixed multiples of the numbers in the table below, and they might vary with the random choices). The default choice of the pricing strategy in that case is *QP_PARTIAL_FILTERED_DANTZIG*.

strategy		runtime in seconds
CGAL::QP_CHOOSE_DEFAULT	—	0.32
CGAL::QP_DANTZIG	—	10.7
CGAL::QP_PARTIAL_DANTZIG	—	3.72
CGAL::QP_BLAND	—	3.65
CGAL::QP_FILTERED_DANTZIG	—	0.43
CGAL::QP_PARTIAL_FILTERED_DANTZIG	—	0.32

We clearly see the effect of filtering: we gain a factor of ten, roughly, compared to the next best non-filtered variant.

10.9.1 $d = 3$, $n = 1,000,000$

The filtering effect is amplified if the points/dimension ratio becomes larger. This is what you might see in dimension three, with one million points.

strategy		runtime in seconds
CGAL::QP_CHOOSE_DEFAULT	—	1.34
CGAL::QP_DANTZIG	—	47.6
CGAL::QP_PARTIAL_DANTZIG	—	15.6
CGAL::QP_BLAND	—	16.02
CGAL::QP_FILTERED_DANTZIG	—	1.89
CGAL::QP_PARTIAL_FILTERED_DANTZIG	—	1.34

In general, if your problem has a high variable/constraint or constraint/variable ratio, then filtering will typically pay off. In such cases, it might be beneficial to encode your problem using input type *double* in order to profit from the filtering (but see the issue discussed in Section 10.8.1).

10.9.2 $d = 100$, $n = 100,000$

Conversely, the filtering effect deteriorates if the points/dimension ratio becomes smaller.

strategy		runtime in seconds
CGAL::QP_CHOOSE_DEFAULT	—	3.05
CGAL::QP_DANTZIG	—	78.4
CGAL::QP_PARTIAL_DANTZIG	—	45.9
CGAL::QP_BLAND	—	33.2
CGAL::QP_FILTERED_DANTZIG	—	3.36
CGAL::QP_PARTIAL_FILTERED_DANTZIG	—	3.06

10.9.3 $d = 500$, $n = 1,000$

If the points/dimension ratio tends to a constant, filtering is no longer a clear winner. The reason is that in this case, the necessary exact calculations with multiprecision numbers dominate the overall runtime.

strategy		runtime in seconds
CGAL::QP_CHOOSE_DEFAULT	—	2.65
CGAL::QP_DANTZIG	—	5.55
CGAL::QP_PARTIAL_DANTZIG	—	5.6
CGAL::QP_BLAND	—	4.46
CGAL::QP_FILTERED_DANTZIG	—	2.65
CGAL::QP_PARTIAL_FILTERED_DANTZIG	—	2.61

In general, if you have a program where the number of variables and the number of constraints have the same order of magnitude, then the saving gained from using the filtered approach is typically small. In such a situation, you should consider switching to a non-filtered variant in order to avoid the rare issue discussed in Section 10.8.1 altogether.

Linear and Quadratic Programming Solver Reference Manual

Kaspar Fischer, Bernd Gärtner, Sven Schönherr, and Frans Wessendorp

10.10 Classified Reference Pages

Concepts

Classes

There is a class that represents the solution of a linear or quadratic program. An instance of this class is returned by any of the solution functions below.

We offer a number of predefined models for the above program concepts. The following two are simultaneously models for all four concepts and are probably the most convenient models; they allow you to construct linear or quadratic programs entry by entry, or from streams in *MPSFormat*. At any time, you can query these programs for linearity and nonnegativity and thus select the appropriate solution function.

Then there are specific models for any of the four program concepts above; these are useful if you want to maintain the program data yourself, since they simply wrap random access iterators over the program data and involve no further copying of data.

 Functions

In case you want to construct a program from complicated iterators (whose types you don't know, or simply don't want to bother with), we provide four makers.

There are four functions to solve a program, one for each program concept.

The solution process can customized by passing an object of the class

Programs can be written to an output stream in *MPSFormat*, using one of the following four functions.

10.11 Alphabetical List of Reference Pages

Concept

QuadraticProgram

Definition

A model of QuadraticProgram describes a convex quadratic program of the form

$$\begin{array}{lll} \text{(QP)} & \text{minimize} & \mathbf{x}^T D\mathbf{x} + \mathbf{c}^T\mathbf{x} + c_0 \\ & \text{subject to} & A\mathbf{x} \gtreqless \mathbf{b}, \\ & & \mathbf{l} \leq \mathbf{x} \leq \mathbf{u} \end{array}$$

in n real variables $\mathbf{x} = (x_0, \ldots, x_{n-1})$. Here,

- A is an $m \times n$ matrix (the constraint matrix),
- $\mathbf{b}$ is an m-dimensional vector (the right-hand side),
- $\gtreqless$ is an m-dimensional vector of relations from $\{\leq, =, \geq\}$,
- $\mathbf{l}$ is an n-dimensional vector of lower bounds for $\mathbf{x}$, where $l_j \in \mathbb{R} \cup \{-\infty\}$ for all j
- $\mathbf{u}$ is an n-dimensional vector of upper bounds for $\mathbf{x}$, where $u_j \in \mathbb{R} \cup \{\infty\}$ for all j
- D is a symmetric positive-semidefinite $n \times n$ matrix (the quadratic objective function),
- $\mathbf{c}$ is an n-dimensional vector (the linear objective function), and
- c_0 is a constant.

The description is given by appropriate *random-access* iterators over the program data, see below. The program therefore comes in *dense* representation which includes zero entries.

Has Models

Quadratic_program<NT>
Quadratic_program_from_mps<NT>
Quadratic_program_from_iterators<A_it, B_it, R_it, FL_it, L_it, FU_it, U_it, D_it, C_it>

Types

QuadraticProgram:: A_iterator	A random access iterator type to go columnwise over the constraint matrix A. The value type is a random access iterator type for an individual column that goes over the entries in that column.
QuadraticProgram:: B_iterator	A random access iterator type to go over the entries of the right-hand side $\mathbf{b}$.
QuadraticProgram:: R_iterator	A random access iterator type to go over the relations $\gtreqless$. The value type of *R_iterator* is *CGAL::Comparison_result*.

QuadraticProgram:: FL_iterator — A random access iterator type to go over the existence (finiteness) of the lower bounds $l_j, j = 0,\ldots,n-1$. The value type of *FL_iterator* is *bool*.

QuadraticProgram:: L_iterator — A random acess iterator type to go over the entries of the lower bound vector **l**.

QuadraticProgram:: UL_iterator — A random access iterator type to go over the existence (finiteness) of the upper bounds $u_j, j = 0,\ldots,n-1$. The value type of *UL_iterator* is *bool*.

QuadraticProgram:: U_iterator — A random acess iterator type to go over the entries of the upper bound vector **u**.

QuadraticProgram:: D_iterator — A random access iterator type to go rowwise over the matrix $2D$. The value type is a random access iterator type for an individual row that goes over the entries in that row, up to (and including) the entry on the main diagonal.

QuadraticProgram:: C_iterator — A random access iterator type to go over the entries of the linear objective function vector c.

Operations

int	*qp.get_n() const*	returns the number n of variables (number of columns of A) in *qp*.
int	*qp.get_m() const*	returns the number m of constraints (number of rows of A) in *qp*.
A_iterator	*qp.get_a() const*	returns an iterator over the columns of A. The corresponding past-the-end iterator is *get_a()+get_n()*. For $j = 0,\ldots,n-1$, $*(get\_a()+j)$ is a random access iterator for column j.
B_iterator	*qp.get_b() const*	returns an iterator over the entries of **b**. The corresponding past-the-end iterator is *get_b()+get_m()*.
R_iterator	*qp.get_r() const*	returns an iterator over the entries of $\gtreqless$. The corresponding past-the-end iterator is *get_r()+get_m()*. The value *CGAL::SMALLER* stands for $\leq$, *CGAL::EQUAL* stands for $=$, and *CGAL::LARGER* stands for $\geq$.
FL_iterator	*qp.get_fl() const*	returns an iterator over the existence of the lower bounds $l_j, j = 0,\ldots,n-1$. The corresponding past-the-end iterator is *get_fl()+get_n()*. If **(get_fl()+j)* has value $true$, the variable x_j has a lower bound given by **(get_l()+j)*, otherwise it has no lower bound.
L_iterator	*qp.get_l() const*	returns an iterator over the entries of **l**. The corresponding past-the-end iterator is *get_l()+get_n()*. If **(get_fl()+j)* has value $false$, the value **(get_l()+j)* is not accessed. *Precondition*: if both **(get_fl()+j)* and **(get_fu()+j)* have value $true$, then $*(get\_l()+j) \leq *(get\_u()+j)$

FU_iterator	*qp.get_fu() const*	returns an iterator over the existence of the upper bounds $u_j, j = 0, \ldots, n-1$. The corresponding past-the-end iterator is *get_fu()+get_n()*. If **(get_fu()+j)* has value $true$, the variable x_j has an upper bound given by **(get_u()+j)*, otherwise it has no upper bound.
L_iterator	*qp.get_u() const*	returns an iterator over the entries of **u**. The corresponding past-the-end iterator is *get_u()+get_n()*. If **(get_fu()+j)* has value $false$, the value **(get_u()+j)* is not accessed. *Precondition*: if both **(get_fl()+j)* and **(get_fu()+j)* have value $true$, then $*(get\_l()+j) \leq *(get\_u()+j)$
D_iterator	*qp.get_d() const*	returns an iterator over the rows of $2D$. The corresponding past-the-end iterator is *get_d()+get_n()*. For $i = 0, \ldots, n-1$, $*(get\_d()+i)$ is a random access iterator for the entries in row i *below or on the diagonal*. The valid range of this iterator is guaranteed to have length $i+1$ but not more. Values to the right of the diagonal are deduced from the symmetry requirement on D.
C_iterator	*qp.get_c() const*	returns an iterator over the entries of **c**. The corresponding past-the-end iterator is *get_c()+get_n()*.

std::iterator_traits<C_iterator>::value_type

	qp.get_c0() const	returns the constant term c_0 of the objective function.

Requirements

The value types of all iterator types (nested iterator types, respectively, for *A_iterator* and *D_iterator*) must be convertible to some common *IntegralDomain* ET.

See Also

The models

Quadratic_program<NT>
Quadratic_program_from_mps<NT>
Quadratic_program_from_iterators<A_it, B_it, R_it, FL_it, L_it, FU_it, U_it, D_it, C_it>

and the other conepts

NonnegativeQuadraticProgramInterface
LinearProgramInterface
NonnegativeLinearProgramInterface

NonnegativeQuadraticProgram

Concept

Definition

A model of NonnegativeQuadraticProgram describes a convex quadratic program of the form

$$\begin{array}{lll} \text{(QP)} & \text{minimize} & \mathbf{x}^T D\mathbf{x} + \mathbf{c}^T\mathbf{x} + c_0 \\ & \text{subject to} & A\mathbf{x} \gtreqless \mathbf{b}, \\ & & \mathbf{x} \geq 0 \end{array}$$

in n real variables $\mathbf{x} = (x_0, \ldots, x_{n-1})$. Here,

- A is an $m \times n$ matrix (the constraint matrix),
- $\mathbf{b}$ is an m-dimensional vector (the right-hand side),
- $\gtreqless$ is an m-dimensional vector of relations from $\{\leq, =, \geq\}$,
- D is a symmetric positive-semidefinite $n \times n$ matrix (the quadratic objective function),
- $\mathbf{c}$ is an n-dimensional vector (the linear objective function), and
- c_0 is a constant.

The description is given by appropriate *random-access* iterators over the program data, see below. The program therefore comes in *dense* representation which includes zero entries.

Has Models

Quadratic_program<NT>
Quadratic_program_from_mps<NT>
Nonnegative_quadratic_program_from_iterators<A_it, B_it, R_it, FL_it, L_it, FU_it, U_it, D_it, C_it>

Types

NonnegativeQuadraticProgram:: A_iterator — A random access iterator type to go columnwise over the constraint matrix A. The value type is a random access iterator type for an individual column that goes over the entries in that column.

NonnegativeQuadraticProgram:: B_iterator — A random access iterator type to go over the entries of the right-hand side $\mathbf{b}$.

NonnegativeQuadraticProgram:: R_iterator — A random access iterator type to go over the relations $\gtreqless$. The value type of *R_iterator* is *CGAL::Comparison_result*.

NonnegativeQuadraticProgram:: D_iterator — A random access iterator type to go rowwise over the matrix $2D$. The value type is a random access iterator type for an individual row that goes over the entries in that row, up to (and including) the entry on the main diagonal.

NonnegativeQuadraticProgram:: C_iterator		A random access iterator type to go over the entries of the linear objective function vector c.

Operations

int	*qp.get_n() const*	returns the number n of variables (number of columns of A) in qp.
int	*qp.get_m() const*	returns the number m of constraints (number of rows of A) in qp.
A_iterator	*qp.get_a() const*	returns an iterator over the columns of A. The corresponding past-the-end iterator is *get_a()+get_n()*. For $j = 0, \ldots, n-1$, $*(get\_a() + j)$ is a random access iterator for column j.
B_iterator	*qp.get_b() const*	returns an iterator over the entries of **b**. The corresponding past-the-end iterator is *get_b()+get_m()*.
R_iterator	*qp.get_r() const*	returns an iterator over the entries of $\gtreqless$. The corresponding past-the-end iterator is *get_r()+get_m()*. The value *CGAL::SMALLER* stands for $\leq$, *CGAL::EQUAL* stands for $=$, and *CGAL::LARGER* stands for $\geq$.
D_iterator	*qp.get_d() const*	returns an iterator over the rows of $2D$. The corresponding past-the-end iterator is *get_d()+get_n()*. For $i = 0, \ldots, n-1$, $*(get\_d() + i)$ is a random access iterator for the entries in row i *below or on the diagonal*. The valid range of this iterator is guaranteed to have length $i+1$ but not more. Values to the right of the diagonal are deduced from the symmetry requirement on D.
C_iterator	*qp.get_c() const*	returns an iterator over the entries of **c**. The corresponding past-the-end iterator is *get_c()+get_n()*.
std::iterator_traits<C_iterator>::value_type		
	qp.get_c0() const	returns the constant term c_0 of the objective function.

Requirements

The value types of all iterator types (nested iterator types, respectively, for *A_iterator* and *D_iterator*) must be convertible to some common *IntegralDomain ET*.

See Also

The models

Quadratic_program<NT>
Quadratic_program_from_mps<NT>
Nonnegative_quadratic_program_from_iterators<A_it, B_it, R_it, D_it, C_it> and the other concepts

QuadraticProgram
LinearProgram
NonnegativeLinearProgram

LinearProgram

Definition

A model of LinearProgram describes a linear program of the form

$$\begin{array}{lll} \text{(QP)} & \text{minimize} & \mathbf{c}^T\mathbf{x}+c_0 \\ & \text{subject to} & A\mathbf{x} \gtreqless \mathbf{b}, \\ & & \mathbf{l} \leq \mathbf{x} \leq \mathbf{u} \end{array}$$

in n real variables $\mathbf{x}=(x_0,\ldots,x_{n-1})$. Here,

- A is an $m\times n$ matrix (the constraint matrix),
- $\mathbf{b}$ is an m-dimensional vector (the right-hand side),
- $\gtreqless$ is an m-dimensional vector of relations from $\{\leq,=,\geq\}$,
- $\mathbf{l}$ is an n-dimensional vector of lower bounds for $\mathbf{x}$, where $l_j \in \mathbb{R}\cup\{-\infty\}$ for all j
- $\mathbf{u}$ is an n-dimensional vector of upper bounds for $\mathbf{x}$, where $u_j \in \mathbb{R}\cup\{\infty\}$ for all j
- $\mathbf{c}$ is an n-dimensional vector (the linear objective function), and
- c_0 is a constant.

The description is given by appropriate *random-access* iterators over the program data, see below. The program therefore comes in *dense* representation which includes zero entries.

Has Models

Quadratic_program<NT>
Quadratic_program_from_mps<NT>
Linear_program_from_iterators<A_it, B_it, R_it, FL_it, L_it, FU_it, U_it, D_it, C_it>

Types

LinearProgram:: A_iterator — A random access iterator type to go columnwise over the constraint matrix A. The value type is a random access iterator type for an individual column that goes over the entries in that column.

LinearProgram:: B_iterator — A random access iterator type to go over the entries of the right-hand side $\mathbf{b}$.

LinearProgram:: R_iterator — A random access iterator type to go over the relations $\gtreqless$. The value type of *R_iterator* is *CGAL::Comparison_result*.

LinearProgram:: FL_iterator — A random access iterator type to go over the existence (finiteness) of the lower bounds $l_j, j=0,\ldots,n-1$. The value type of *FL_iterator* is *bool*.

LinearProgram:: L_iterator		A random acess iterator type to go over the entries of the lower bound vector **l**.
LinearProgram:: UL_iterator		A random access iterator type to go over the existence (finiteness) of the upper bounds $u_j, j = 0, \ldots, n-1$. The value type of *UL_iterator* is *bool*.
LinearProgram:: U_iterator		A random acess iterator type to go over the entries of the upper bound vector **u**.
LinearProgram:: C_iterator		A random access iterator type to go over the entries of the linear objective function vector c.

Operations

int	*lp.get_n() const*	returns the number n of variables (number of columns of A) in *lp*.
int	*lp.get_m() const*	returns the number m of constraints (number of rows of A) in *lp*.
A_iterator	*lp.get_a() const*	returns an iterator over the columns of A. The corresponding past-the-end iterator is *get_a()+get_n()*. For $j = 0, \ldots, n-1$, $*(get\_a() + j)$ is a random access iterator for column j.
B_iterator	*lp.get_b() const*	returns an iterator over the entries of **b**. The corresponding past-the-end iterator is *get_b()+get_m()*.
R_iterator	*lp.get_r() const*	returns an iterator over the entries of $\gtreqless$. The corresponding past-the-end iterator is *get_r()+get_m()*. The value *CGAL::SMALLER* stands for $\leq$, *CGAL::EQUAL* stands for $=$, and *CGAL::LARGER* stands for $\geq$.
FL_iterator	*lp.get_fl() const*	returns an iterator over the existence of the lower bounds $l_j, j = 0, \ldots, n-1$. The corresponding past-the-end iterator is *get_fl()+get_n()*. If **(get_fl()+j)* has value $true$, the variable x_j has a lower bound given by **(get_l()+j)*, otherwise it has no lower bound.
L_iterator	*lp.get_l() const*	returns an iterator over the entries of **l**. The corresponding past-the-end iterator is *get_l()+get_n()*. If **(get_fl()+j)* has value $false$, the value **(get_l()+j)* is not accessed. *Precondition*: if both **(get_fl()+j)* and **(get_fu()+j)* have value $true$, then $*(get\_l() + j) \leq *(get\_u() + j)$
FU_iterator	*lp.get_fu() const*	returns an iterator over the existence of the upper bounds $u_j, j = 0, \ldots, n-1$. The corresponding past-the-end iterator is *get_fu()+get_n()*. If **(get_fu()+j)* has value $true$, the variable x_j has an upper bound given by **(get_u()+j)*, otherwise it has no upper bound.

L_iterator	*lp.get_u() const*	returns an iterator over the entries of **u**. The corresponding past-the-end iterator is *get_u()+get_n()*. If **(get_fu()+j)* has value $false$, the value **(get_u()+j)* is not accessed. *Precondition*: if both **(get_fl()+j)* and **(get_fu()+j)* have value $true$, then $*(get\_l()+j) \leq *(get\_u()+j)$
C_iterator	*lp.get_c() const*	returns an iterator over the entries of **c**. The corresponding past-the-end iterator is *get_c()+get_n()*.
std::iterator_traits<C_iterator>::value_type		
	lp.get_c0() const	returns the constant term c_0 of the objective function.

See Also

The models

Quadratic_program<NT>
Quadratic_program_from_mps<NT>
Linear_program_from_iterators<A_it, B_it, R_it, FL_it, L_it, FU_it, U_it, C_it>

and the other concepts

QuadraticProgram
NonnegativeQuadraticProgram
NonnegativeLinearProgram

NonnegativeLinearProgram

Definition

A model of NonnegativeLinearProgram describes a linear program of the form

$$\begin{array}{lll} \text{(QP)} & \text{minimize} & \mathbf{c}^T\mathbf{x} + c_0 \\ & \text{subject to} & A\mathbf{x} \gtreqless \mathbf{b}, \\ & & \mathbf{x} \geq 0 \end{array}$$

in n real variables $\mathbf{x} = (x_0, \ldots, x_{n-1})$. Here,

- A is an $m \times n$ matrix (the constraint matrix),
- $\mathbf{b}$ is an m-dimensional vector (the right-hand side),
- $\gtreqless$ is an m-dimensional vector of relations from $\{\leq, =, \geq\}$,
- $\mathbf{c}$ is an n-dimensional vector (the linear objective function), and
- c_0 is a constant.

The description is given by appropriate *random-access* iterators over the program data, see below. The program therefore comes in *dense* representation which includes zero entries.

Has Models

Quadratic_program<NT>
Quadratic_program_from_mps<NT>
Nonnegative_linear_program_from_iterators<A_it, B_it, R_it, FL_it, L_it, FU_it, U_it, D_it, C_it>

Types

NonnegativeLinearProgram:: A_iterator
A random access iterator type to go columnwise over the constraint matrix A. The value type is a random access iterator type for an individual column that goes over the entries in that column.

NonnegativeLinearProgram:: B_iterator
A random access iterator type to go over the entries of the right-hand side $\mathbf{b}$.

NonnegativeLinearProgram:: R_iterator
A random access iterator type to go over the relations $\gtreqless$. The value type of *R_iterator* is *CGAL::Comparison_result*.

NonnegativeLinearProgram:: C_iterator
A random access iterator type to go over the entries of the linear objective function vector c.

Operations

int	*lp.get_n() const*	returns the number n of variables (number of columns of A) in *lp*.
int	*lp.get_m() const*	returns the number m of constraints (number of rows of A) in *lp*.
A_iterator	*lp.get_a() const*	returns an iterator over the columns of A. The corresponding past-the-end iterator is *get_a()+get_n()*. For $j = 0, \ldots, n-1$, $*(get\_a() + j)$ is a random access iterator for column j.
B_iterator	*lp.get_b() const*	returns an iterator over the entries of **b**. The corresponding past-the-end iterator is *get_b()+get_m()*.
R_iterator	*lp.get_r() const*	returns an iterator over the entries of $\gtreqless$. The corresponding past-the-end iterator is *get_r()+get_m()*. The value *CGAL::SMALLER* stands for $\leq$, *CGAL::EQUAL* stands for $=$, and *CGAL::LARGER* stands for $\geq$.
C_iterator	*lp.get_c() const*	returns an iterator over the entries of **c**. The corresponding past-the-end iterator is *get_c()+get_n()*.
std::iterator_traits<C_iterator>::value_type	*lp.get_c0() const*	returns the constant term c_0 of the objective function.

Requirements

The value types of all iterator types (nested iterator types, respectively, for *A_iterator*) must be convertible to some common *IntegralDomain ET*.

See Also

The models

Quadratic_program<NT>
Quadratic_program_from_mps<NT>
Nonnegative_linear_program_from_iterators<A_it, B_it, R_it, C_it>

and the other concepts

QuadraticProgram
LinearProgram
NonnegativeQuadraticProgram

MPSFormat

MPS is a commonly used file format for storing linear and quadratic programs according to the concepts *QuadraticProgram*, *LinearProgram*, *NonnegativeQuadraticProgram*, and *NonnegativeLinearProgram*, see also http://en.wikipedia.org/wiki/MPS_(format).

CGAL supports a large subset of this format, but there are MPS files around that we cannot read (for example, files that encode integrality constraints on the variables). Also, there might be some other MPS-based solvers that will not be able to read the MPS files written by CGAL, since we do not strictly adhere to the very rigid layout requirements of the original MPS format.

Let's look at an example first. The quadratic program

$$\begin{array}{lrcl}
\text{minimize} & x^2+4(y-4)^2 & (= & x^2+4y^2-32y+64) \\
\text{subject to} & x+y & \leq & 7 \\
 & -x+2y & \leq & 4 \\
 & x & \geq & 0 \\
 & y & \geq & 0 \\
 & y & \leq & 4
\end{array}$$

has the following description in MPS format.

```
NAME first_qp
ROWS
  N obj
  L c0
  L c1
COLUMNS
  x0  c0  1
  x0  c1  -1
  x1  obj  -32
  x1  c0  1
  x1  c1  2
RHS
  rhs obj -64
  rhs c0  7
  rhs c1  4
BOUNDS
  UP  BND  x1  4
QMATRIX
  x0  x0  2
  x1  x1  8
ENDATA
```

Here comes a semiformal description of the format in general.

NAME section

This (mandatory) section consists of a single line starting with `NAME`. Everything starting from the first non-whitespace after that until the end of the line constitutes the name of the problem.

ROWS section

In the (mandatory) `ROW` section, you find one line for every constraint, where the letter `L` indicates relation $\leq$, letter `G` stands for $\geq$, and `E` for $=$. In addition, there is a row for the linear objective function (indicated by letter `N`). In that section, names are asigned to the constraints (here: `c0, c1`) and the objective function (here: `obj`). An MPS file may encode several linear objective functions by using several rows starting with `N`, but we ignore all but the first.

COLUMNS section

The (mandatory) `COLUMNS` section encodes the constraint matrix A and the linear objective function vector c. Every line consists of one or two sequences of three tokens $j i val$, where j is the name of a variable (here, we have variables `x0,x1`), i is the name of a constraint or the objective function, and val is the value A_{ij} (if i names a constraint), or c_j (if i names the linear objective function). Values for pairs (i, j) that are not specified in this section default to 0. Otherwise, for every pair (i, j), the *last* specified value determines A_{ij} or c_j.

RHS section

This (mandatory) section encodes the right-hand side vector b and the constant term c_0 in the objective function. The first token in every line is an identifier (here: `rhs`). An MPS file may encode several right-hand sides b by using several such identifiers, but we ignore all lines having an identifier different from that of the first line.

The right-hand side identifier is succeeded by one or two sequences of tokens $i val$, where i names a constraint or the linear objective function, and val specifies the value b_i (if i names a constraint), or $-c_0$ (if i names the linear objective function). Values that are not specified in this section default to 0. Otherwise, for every i, the *last* specified value determines b_i or $-c_0$.

BOUNDS section

This (optional) section encodes the lower and upper bound vectors l and u for the variables. The default bounds for any variable x_j are $0 \leq x_j \leq \infty$; the `BOUNDS` section is used to override these defaults. In particular, if there is no `BOUNDS` section, the program is nonnegative and actually a model of the concept *NonnegativeQuadraticProgram* or *NonnegativeLinearProgram*.

The first token in every line is succeeded by an (optional) identifier (here: `BND`). An MPS file may encode several bound vectors l and u by using several such identifiers, but we ignore all lines having an identifier different from that of the first line. The first token t itself determines the type of the bound, and the token j after the bound identifier names the variable to which the bound applies In case of bound types `FX`, `LO`, and `UP`, there is another token *val* that specifices the bound value. Here is how bound type and value determine a bound for variable x_j. There may be several bound specifications for a single variable, and they are processed in order of appearance.

bound type	resulting bound
FX	$x_j = val$ (x_j becomes a fixed variable)
LO	$x_j \geq val$ (upper bound remains unchanged)
UP	$x_j \leq val$ (lower bound remains unchanged, except if $val < 0$; then, a zero lower bound is reset to $-\infty$)
FR	$-\infty \leq x_j \leq \infty$ (previous bounds are discarded)
MI	$x_j \geq -\infty$ (upper bound remains unchanged)
PL	$x_j \leq \infty$ (lower bound remains unchanged)

QMATRIX / QUADOBJ / DMATRIX section

This (optional) section encodes the quadratic objective function matrix D. Every line is a sequence $i\,j\,val$ of three tokens, where both i and j name variables, and *val* is the value $2D_{i,j}$ (in case of `QMATRIX` or `QUADOBJ`), or D_{ij} (in case of `DMATRIX`).

In case of `QMATRIX` and `DMATRIX`, *all* nonzero entries must be specified: if there is a line $i\,j\,val$, then there must also be a line $j\,i\,val$, since D is required to be symmetric. In case of `QUADOBJ`, only the entries of $2D$ on or below the diagonal must be specified, entries above the diagonal are deduced from symmetry. It is not allowed to specify two or more *different* nonzero values for an unordered pair $\{i, j\}$.

If this section is missing or does not contain nonzero values, the program is a model of the concept *LinearProgram*.

Miscellaneous

Our MPS format also supports an (optional) `RANGES` section, but we don't explain this here.

See Also

Quadratic_program_from_mps<NT>

Class

CGAL::Quadratic_program_solution<ET>

#include <CGAL/QP_solution.h>

Definition

An object of class *Quadratic_program_solution<ET>* represents the solution of a linear or convex quadratic program of the general form

$$
\begin{array}{lll}
\text{(QP)} & \text{minimize} & \mathbf{x}^T D\mathbf{x} + \mathbf{c}^T\mathbf{x} + c_0 \\
 & \text{subject to} & A\mathbf{x} \gtreqless \mathbf{b}, \\
 & & \mathbf{l} \leq \mathbf{x} \leq \mathbf{u}
\end{array}
$$

in n real variables $\mathbf{x} = (x_0, \ldots, x_{n-1})$.

If $D = 0$, the program is a linear program; if the variable bounds are $\mathbf{x} \geq 0$, we have a nonnegative program. Objects of type *Quadratic_program_solution<ET>* are returned by any of the four functions *solve_quadratic_program*, *solve_linear_program*, *solve_nonnegative_quadratic_program*, and *solve_nonnegative_linear_program*.

Example

QP_solver/first_qp.cpp

Terminology

If there is no $\mathbf{x}$ that satisfies all the (in)equalities, the program is called *infeasible*, otherwise, it is *feasible*, and any $\mathbf{x}$ that satisfies all (in)equalities is called a *feasible solution*.

If the objective function value becomes arbitrarily small over the *feasible region* (the set of feasible solutions), the program is called *unbounded*, and *bounded* otherwise.

Any program that is both feasible and bounded has at least one feasible solution $\mathbf{x}^*$ whose objective function value is not larger than that of any other feasible solution. This is called an *optimal solution*.

Every convex quadratic program (even if it is infeasible or unbounded) has a 'solution' in form of an object of the class *Quadratic_program_solution<ET>*.

Types

Quadratic_program_solution<ET>:: ET	The exact number type that was used to solve the program.
Quadratic_program_solution<ET>:: Variable_value_iterator	An iterator type with value type *Quotient<ET>* to go over the values of the variables in the solution.

Quadratic_program_solution<ET>:: Variable_numerator_iterator

An iterator type with value type *ET* to go over the numerators of the variable values with respect to a common denominator.

Quadratic_program_solution<ET>:: Index_iterator

An iterator type with value type *int* to go over the indices of the basic variables and the basic constraints.

Quadratic_program_solution<ET>:: Optimality_certificate_iterator

An iterator type with value type *Quotient<ET>* to go over an m-vector λ that proves optimality of the solution.

Quadratic_program_solution<ET>:: Optimality_certificate_numerator_iterator

An iterator type with value type *ET* to go over the numerators of the vector λ with respect to a common denominator.

Quadratic_program_solution<ET>:: Infeasibility_certificate_iterator

An iterator type with value type *ET* to go over an m-vector λ that proves infeasibility of the solution.

Quadratic_program_solution<ET>:: Unboundedness_certificate_iterator

An iterator type with value type *ET* to go over an n-vector $\mathbf{w}$ that proves unboundedness of the solution.

Creation

Quadratic_program_solution<ET> sol; — constructs a void instance of *Quadratic_program_solution<ET>* that is associated to no program.

Objects of type *Quadratic_program_solution<ET>* can be copied and assigned.

Objects of type *Quadratic_program_solution<ET>* that are associated to an actual program are returned by any of the four functions *solve_quadratic_program*, *solve_linear_program*, *solve_nonnegative_quadratic_program*, and *solve_nonnegative_linear_program*.

Example

QP_solver/first_qp.cpp

Operations

bool *sol.is_void() const* returns *true* iff *sol* is not associated to a program. The condition !*sol.is_void()* is a precondition for all access methods below.

Solution status

Here are the access methods for the status of the solution.

bool *sol.is_optimal() const* returns *true* iff *sol* is an optimal solution of the associated program.

bool *sol.is_infeasible() const*

returns *true* iff the associated program is infeasible.

bool *sol.is_unbounded() const*

returns *true* iff the associated program is unbounded.

Quadratic_program_status

sol.status() const returns the status of the solution; this is one of the values *QP_OPTIMAL*, *QP_INFEASIBLE*, and *QP_UNBOUNDED*, depending on whether the program asociated to *sol* has an optimal solution, is infeasible, or is unbounded.

int *sol.number_of_iterations() const*

returns the number of iterations that it took to solve the program asociated to *sol*.

Solution values

The actual solution (variable and objective function values) can be accessed as follows.

Quotient<ET> *sol.objective_value() const*

returns the objective function value of *sol*.

ET *sol.objective_value_numerator() const*

returns the numerator of the objective function value of *sol*.

ET *sol.objective_value_denominator() const*

returns the denominator of the objective function value of *sol*.

Variable_value_iterator

sol.variable_values_begin() const

returns a random-access iterator over the values of the variables in *sol*. The value type is *Quotient<ET>*, and the valid iterator range has length *n*.

Variable_value_iterator

sol.variable_values_end() const

returns the corresponding past-the-end iterator.

Variable_numerator_iterator

sol.variable_numerators_begin() const

returns a random-access iterator *it* over the values of the variables in *sol*, with respect to a common denominator of all variables. The value type is *ET*, and the valid iterator range has length *n*.

Variable_numerator_iterator

sol.variable_numerators_end() const

returns the corresponding past-the-end iterator.

ET *sol.variables_common_denominator() const*

returns the common denominator of the variable values as referred to by the previous two methods.

Basic variables and constraints

The solution of a linear or quadratic program distinguishes 'important' variables (the ones not attaining one of their bounds), and 'important' constraints (the ones being satisfied with equality). The following methods grant access to them.

Index_iterator *sol.basic_variable_indices_begin() const*

returns a random access iterator over the indices of the basic variables. The value type is *int*. It is guaranteed that any variable that is not basic in *sol* attains one of its bounds. In particular, if the bounds are of type $\mathbf{x} \geq 0$, all non-basic variables have value 0.

Index_iterator *sol.basic_variable_indices_end() const*

returns the corresponding past-the-end iterator.

int *sol.number_of_basic_variables() const*

returns the number of basic variables, equivalently the length of the range determined by the previous two iterators.

Example

QP_solver/important_variables.cpp

Index_iterator *sol.basic_constraint_indices_begin() const*

returns a random access iterator over the indices of the basic constraints in the system $A\mathbf{x} \gtreqless \mathbf{b}$. The value type is *int*. It is guaranteed that any basic constraint is satisfied with equality. In particular, if the system is of type $A\mathbf{x} = \mathbf{b}$, all constraints are basic.

Index_iterator *sol.basic_constraint_indices_end() const*

returns the corresponding past-the-end iterator.

int *sol.number_of_basic_constraints() const*

returns the number of basic constraint, equivalently the length of the range determined by the previous two iterators.

Example

QP_solver/first_qp_basic_constraints.cpp

Output

template <typename ET>
std::ostream& *std::ostream& out << sol*

writes the status of *sol* to the stream *out*. In case the status is *QP_OPTIMAL*, the optimal objective value and the values of the variables at the optimal solution are output as well. For more detailed information about the solution (like basic variables/constraints) please use the dedicated methods of *Quadratic_program_solution<ET>*.

Validity

The following four methods allow you to check whether *sol* indeed solves the program that you intended to solve. The methods use the certificates described in the advanced section below and thus save you from validating the certificates yourself (if you believe in the correctness of these methods; otherwise, you can look at their implementation to convince yourself).

By passing a suitable option to the solution function, you can make sure that this check is done automatically after the solution of the program, see *Quadratic_program_options*. If the check fails, a logfile is generated that contains the details, and an error message is written to *std::cerr* (see *QP_solver/cycling.cpp* for an example that uses this option).

template <class QuadraticProgram>
bool *sol.solves_quadratic_program(QuadraticProgram qp)*

returns *true* iff *sol* solves the quadratic program *qp*. If the result is *false*, you can get a message that describes the problem, through the method *get_error()*.

QP_solver/first_qp.cpp

template <class LinearProgram>
bool *sol.solves_linear_program(LinearProgram lp)*

returns *true* iff *sol* solves the linear program *lp*. If the result is *false*, you can get a message that describes the problem, through the method *get_error()*.

QP_solver/first_lp.cpp

template <class NonnegativeQuadraticProgram>
bool *sol.solves_nonnegative_quadratic_program(NonnegativeQuadraticProgram qp)*

returns *true* iff *sol* solves the nonnegative quadratic program *qp*. If the result is *false*, you can get a message that describes the problem, through the method *get_error()*.

QP_solver/first_nonnegative_qp.cpp

template <class NonnegativeLinearProgram>
bool *sol.solves_nonnegative_linear_program(NonnegativeLinearProgram lp)*

returns *true* iff *sol* solves the nonnegative linear program *lp*. If the result is *false*, you can get a message that describes the problem, through the method *get_error()*.

QP_solver/first_nonnegative_lp.cpp

bool *sol.is_valid() const* returns *false* iff the validation through one of the previous four functions has failed.

std::string *sol.get_error() const* returns an error message in case any of the previous four validation functions has returned *false*.

advanced

Certificates

A certificate is a vector that admits a simple proof for the correctness of the solution. Any non-void object of *Quadratic_program_solution<ET>* comes with such a certificate.

Lemma 1 (optimality certificate): A feasible n-vector $\mathbf{x}^*$ is an optimal solution of (QP) if an m-vector λ with the following properties exist.

1. if the i-th constraint is of type $\leq$ ($\geq$, respectively), then $\lambda_i \geq 0$ ($\lambda_i \leq 0$, respectively).
2. $\lambda^T(A\mathbf{x}^* - \mathbf{b}) = 0$.
3. $$(\mathbf{c}^T + \lambda^T A + 2\mathbf{x}^{*T} D)_j \quad \begin{array}{ll} \geq 0, & \text{if } x^*_j = l_j < u_j \\ = 0, & \text{if } l_j < x^*_j < u_j \\ \leq 0, & \text{if } l_j < u_j = x^*_j. \end{array}$$

Proof: Let $\mathbf{x}$ be any feasible solution. We need to prove that

$$\mathbf{c}^T\mathbf{x} + \mathbf{x}^T D\mathbf{x} \geq \mathbf{c}^T\mathbf{x}^* + \mathbf{x}^{*T} D\mathbf{x}^*.$$

For this, we argue as follows.

$$\begin{array}{rcll} \mathbf{c}^T\mathbf{x} + 2\mathbf{x}^{*T} D\mathbf{x} & \geq & \mathbf{c}^T\mathbf{x} + 2\mathbf{x}^{*T} D\mathbf{x} + \lambda^T(A\mathbf{x} - \mathbf{b}) & (\text{by } A\mathbf{x} \gtreqless \mathbf{b} \text{ and 1.}) \\ & = & (\mathbf{c}^T + \lambda^T A + 2\mathbf{x}^{*T} D)\mathbf{x} - \lambda^T b & \\ & \geq & (\mathbf{c}^T + \lambda^T A + 2\mathbf{x}^{*T} D)\mathbf{x}^* - \lambda^T b & (\text{by } \mathbf{l} \leq \mathbf{x} \leq \mathbf{u} \text{ and 3.}) \\ & = & \mathbf{c}^T\mathbf{x}^* + 2\mathbf{x}^{*T} D\mathbf{x}^* & (\text{by 2.}) \end{array}$$

After adding $\mathbf{x}^T D\mathbf{x} - \mathbf{x}^T D\mathbf{x} - \mathbf{x}^{*T} D\mathbf{x}^* = -\mathbf{x}^{*T} D\mathbf{x}^*$ to both sides of this inequality, we get

$$\mathbf{c}^T\mathbf{x} + \mathbf{x}^T D\mathbf{x} - (\mathbf{x} - \mathbf{x}^*)^T D(\mathbf{x} - \mathbf{x}^*) \geq \mathbf{c}^T\mathbf{x}^* + \mathbf{x}^{*T} D\mathbf{x}^*,$$

and since D is positive semidefinite, we have $(\mathbf{x} - \mathbf{x}^*)^T D(\mathbf{x} - \mathbf{x}^*) \geq 0$ and the lemma follows.

Optimality_certificate_iterator

sol.optimality_certifcate_begin() const

returns a random access iterator over the optimality certificate λ as given in Lemma 1, with respect to the solution $\mathbf{x}^*$ obtained from *sol.variable_values_begin()*. The value type is *Quotient<ET>*, and the valid iterator range has length m.
Precondition: *sol.is_optimal()*

Optimality_certificate_iterator

sol.optimality_certificate_end() const

returns the corresponding past-the-end iterator.

Optimality_certificate_numerator_iterator

sol.optimality_certifcate_numerators_begin() const

returns a random access iterator over the numerator values of the optimality certificate λ, with respect to the common denominator returned by *sol.variables_common_denominator()*. The value type is *ET*, and the valid iterator range has length m.

Optimality_certificate_numerator_iterator

sol.optimality_certificate_numerators_end() const

returns the corresponding past-the-end iterator.

Example

QP_solver/optimality_certificate.cpp

Lemma 2 (infeasibility certificate): The program (QP) is infeasible if an m-vector λ with the following properties exist.

1. if the i-th constraint is of type $\leq$ ($\geq$, respectively), then $\lambda_i \geq 0$ ($\lambda_i \leq 0$, respectively).
2. $$\lambda^T A_j \quad \begin{array}{ll} \geq 0 & \text{if } u_j = \infty \\ \leq 0 & \text{if } l_j = -\infty. \end{array}$$
3. $$\lambda^T \mathbf{b} \quad < \quad \sum_{j:\lambda^T A_j < 0} \lambda^T A_j u_j \quad + \quad \sum_{j:\lambda^T A_j > 0} \lambda^T A_j l_j.$$

Proof: Let us assume for the purpose of obtaining a contradiction that there is a feasible solution $\mathbf{x}$. Then we get

$$\begin{array}{rclll} 0 & \geq & \lambda^T(A\mathbf{x}-\mathbf{b}) & & \text{(by } A\mathbf{x} \gtreqless \mathbf{b} \text{ and 1.)} \\ & = & \sum_{j:\lambda^T A_j<0} \lambda^T A_j x_j & + \quad \sum_{j:\lambda^T A_j>0} \lambda^T A_j x_j - \lambda^T \mathbf{b} & \\ & \geq & \sum_{j:\lambda^T A_j<0} \lambda^T A_j u_j & + \quad \sum_{j:\lambda^T A_j>0} \lambda^T A_j l_j - \lambda^T \mathbf{b} & \text{(by } \mathbf{l} \leq \mathbf{x} \leq \mathbf{u} \text{ and 2.)} \\ & > & 0 & & \text{(by 3.),} \end{array}$$

and this is the desired contradiction $0 > 0$.

Infeasibility_certificate_iterator

sol.infeasibility_certificate_begin() const

returns a random access iterator over the infeasibility certificate λ as given in Lemma 2. The value type is *ET*, and the valid iterator range has length m.
Precondition: *sol.is_infeasible()*

Infeasibility_certificate_iterator

sol.infeasibility_certificate_end() const

returns the corresponding past-the-end iterator.

Example

QP_solver/infeasibility_certificate.cpp

Lemma 3 (unboundedness certificate:) Let $\mathbf{x}^*$ be a feasible solution of (QP). The program (QP) is unbounded if an n-vector $\mathbf{w}$ with the following properties exist.

1. if the i-th constraint is of type $\leq$ ($\geq, =$, respectively), then $(A\mathbf{w})_i \leq 0$ ($(A\mathbf{w})_i \geq 0, (A\mathbf{w})_i = 0$, respectively).
2. $$w_j \quad \begin{array}{ll} \geq 0 & \text{if } l_j \text{ is finite} \\ \leq 0 & \text{if } u_j \text{ is finite.} \end{array}$$
3. $\mathbf{w}^T D\mathbf{w} = 0$ and $(\mathbf{c}^T + 2\mathbf{x}^{*T} D)\mathbf{w} < 0$.

The vector $\mathbf{w}$ is called an *unbounded direction*.

Proof: For a real number t, consider the vector $\mathbf{x}(t) := \mathbf{x}^* + t\mathbf{w}$. By 1. and 2., $\mathbf{x}(t)$ is feasible for all $t \geq 0$. The objective function value of $\mathbf{x}(t)$ is

$$\begin{aligned} \mathbf{c}^T\mathbf{x}(t) + \mathbf{x}(t)^T D\mathbf{x}(t) &= \mathbf{c}^T\mathbf{x}^* + t\mathbf{c}^T\mathbf{w} + \mathbf{x}^{*T} D\mathbf{x}^* + 2t\mathbf{x}^{*T} D\mathbf{w} + t^2\mathbf{w}^T D\mathbf{w} \\ &= \mathbf{c}^T\mathbf{x}^* + \mathbf{x}^{*T} D\mathbf{x}^* + t(\mathbf{c}^T + 2\mathbf{x}^{*T} D)\mathbf{w} + t^2\mathbf{w}^T D\mathbf{w}. \end{aligned}$$

By condition 3., this tends to $-\infty$ for $t \to \infty$, so the problem is indeed unbounded.

Unboundedness_certificate_iterator

sol.unboundedness_certificate_begin() const

returns a random acess iterator over the unbounded direction $\mathbf{w}$ as given in Lemma 3,with respect to the solution $\mathbf{x}^*$ obtained from *sol.variable_values_begin()*. The value type is *ET*, and the valid iterator range has length n.
Precondition: *sol.is_unbounded()*

Unboundedness_certificate_iterator

sol.unboundedness_certificate_end()

returns the corresponding past-the-end iterator.

Example

QP_solver/unboundedness_certificate.cpp

advanced

See Also

The program concepts

QuadraticProgram . . . . page 500
LinearProgram . . . . page 506
NonnegativeQuadraticProgram . . . . page 503
NonnegativeLinearProgram . . . . page 509

and the functions that compute objects of class *Quadratic_program_solution<ET>* from models of these concepts:

solve_quadratic_program . . . . page **??**
solve_linear_program . . . . page **??**
solve_nonnegative_quadratic_program . . . . page **??**
solve_nonnegative_linear_program . . . . page **??**

Class

CGAL::Quadratic_program_status

#include <CGAL/QP_solution.h>

Definition

This is an enumeration type containing the values *QP_OPTIMAL*, *QP_INFEASIBLE*, and *QP_UNBOUNDED*. It indicates the status of a linear or quadratic program solution as represented by an object of type *Quadratic_program_solution<ET>*.

See Also

Quadratic_program_solution<ET>

CGAL::Quadratic_program<NT>

#include <CGAL/QP_models.h>

Class

Definition

An object of class *Quadratic_program<NT>* describes a convex quadratic program of the form

$$\begin{array}{lll}\text{(QP)} & \text{minimize} & \mathbf{x}^T D\mathbf{x}+\mathbf{c}^T\mathbf{x}+c_0 \\ & \text{subject to} & A\mathbf{x} \gtreqless \mathbf{b}, \\ & & \mathbf{l} \leq \mathbf{x} \leq \mathbf{u}\end{array}$$

in n real variables $\mathbf{x} = (x_0,\ldots,x_{n-1})$. Here,

- A is an $m \times n$ matrix (the constraint matrix),
- $\mathbf{b}$ is an m-dimensional vector (the right-hand side),
- $\gtreqless$ is an m-dimensional vector of relations from $\{\leq,=,\geq\}$,
- $\mathbf{l}$ is an n-dimensional vector of lower bounds for $\mathbf{x}$, where $l_j \in \mathbb{R} \cup \{-\infty\}$ for all j
- $\mathbf{u}$ is an n-dimensional vector of upper bounds for $\mathbf{x}$, where $u_j \in \mathbb{R} \cup \{\infty\}$ for all j
- D is a symmetric positive-semidefinite $n \times n$ matrix (the quadratic objective function),
- $\mathbf{c}$ is an n-dimensional vector (the linear objective function), and
- c_0 is a constant.

If $D = 0$, the program is a linear program; if the variable bounds are $x \geq 0$, we have a nonnegative program.

This class allows you to build your program entry by entry, using the set-methods below.

If you only need to wrap existing (random-access) iterators over your own data, then you may use any of the four models *Quadratic_program_from_iterators<A_it, B_it, R_it, FL_it, L_it, FU_it, U_it, D_it, C_it>*, *Linear_program_from_iterators<A_it, B_it, R_it, FL_it, L_it, FU_it, U_it, C_it>*, *Nonnegative_quadratic_program_from_iterators<A_it, B_it, R_it, D_it, C_it>*, and *Nonnegative_linear_program_from_iterators<A_it, B_it, R_it, C_it>*.

If you want to read a quadratic program in *MPSFormat* from a stream, please use the model *Quadratic_program_from_mps<NT>*.

Is Model for the Concepts

QuadraticProgram
LinearProgram
NonnegativeQuadraticProgram
NonnegativeLinearProgram

Types

Quadratic_program<NT>:: NT	The number type of the program entries.

Creation

Quadratic_program<NT> qp(CGAL::Comparison_result default_r = CGAL::EQUAL, bool default_fl = true, NT default_l = 0, bool default_fu = false, NT default_u = 0)

constructs a quadratic program with no variables and no constraints, ready for data to be added. Unless relations are explicitly set, they will be of type *default_r*. Unless bounds are explicitly set, they will be as specified by *default_fl* (finite lower bound?), *default_l* (lower bound value if lower bound is finite), *default_fu* (finite upper bound?), and *default_l* (upper bound value if upper bound is finite). If all parameters take their default values, we thus get equality constraints and bounds $x \geq 0$ by default. Numerical entries that are not explicitly set will default to 0.
Precondition: if *default_fl* = *default_fu* = *true*, then *default_l* $\leq$ *default_u*

Operations

bool *qp.is_linear() const*

returns *true* if and only if *qp* is a linear program.

bool *qp.is_nonnegative() const*

returns *true* if and only if *qp* is a nonnegative program.

void *qp.set_a(int j, int i, NT val)*

sets the entry A_{ij} in column j and row i of the constraint matrix A of *qp* to *val*. An existing entry is overwritten. *qp* is enlarged if necessary to accomodate this entry.

void *qp.set_b(int i, NT val)*

sets the entry b_i of *qp* to *val*. An existing entry is overwritten. *qp* is enlarged if necessary to accomodate this entry.

void *qp.set_r(int i, CGAL::Comparison_result rel)*

sets the entry $\gtreqless_i$ of *qp* to *rel*. *CGAL::SMALLER* means that the i-th constraint is of type "$\leq$", *CGAL::EQUAL* means "$=$", and *CGAL::LARGER* encodes "$\geq$". An existing entry is overwritten. *qp* is enlarged if necessary to accomodate this entry.

void *qp.set_l(int j, bool is_finite, NT val = NT(0))*

if *is_finite*, this sets the entry l_j of *qp* to *val*, otherwise it sets l_j to $-\infty$. An existing entry is overwritten. *qp* is enlarged if necessary to accomodate this entry.

void *qp.set_u(int j, bool is_finite, NT val = NT(0))*

if *is_finite*, this sets the entry u_j of *qp* to *val*, otherwise it sets u_j to ∞. An existing entry is overwritten. *qp* is enlarged if necessary to accomodate this entry.

void *qp.set_c(int j, NT val)*

sets the entry c_j of *qp* to *val*. An existing entry is overwritten. *qp* is enlarged if necessary to accomodate this entry.

void *qp.set_c0(NT val)* sets the entry c_0 of *qp* to *val*. An existing entry is overwritten.

void *qp.set_d(int i, int j, NT val)*

sets the entries $2D_{ij}$ and $2D_{ji}$ of *qp* to *val*. Existing entries are overwritten. *qp* is enlarged if necessary to accomodate these entries.
Precondition: *j <= i*

Example

QP_solver/first_qp.cpp
QP_solver/first_lp.cpp
QP_solver/first_nonnegative_qp.cpp
QP_solver/first_nonnegative_lp.cpp
QP_solver/invert_matrix.cpp **See Also**

Quadratic_program_from_iterators<A_it, B_it, R_it, FL_it, L_it, FU_it, U_it, D_it, C_it>
Linear_program_from_iterators<A_it, B_it, R_it, FL_it, L_it, FU_it, U_it, C_it>
Nonnegative_quadratic_program_from_iterators<A_it, B_it, R_it, D_it, C_it>
Nonnegative_linear_program_from_iterators<A_it, B_it, R_it, C_it>
Quadratic_program_from_mps<NT>

Class

CGAL::Quadratic_program_from_mps<NT>

#include <CGAL/QP_models.h>

Definition

An object of class *Quadratic_program_from_mps<NT>* describes a convex quadratic program of the general form

$$
\begin{array}{lll}
\text{(QP)} & \text{minimize} & \mathbf{x}^T D\mathbf{x} + \mathbf{c}^T\mathbf{x} + c_0 \\
 & \text{subject to} & A\mathbf{x} \gtreqless \mathbf{b}, \\
 & & \mathbf{l} \leq \mathbf{x} \leq \mathbf{u}
\end{array}
$$

in n real variables $\mathbf{x} = (x_0, \ldots, x_{n-1})$. Here,

- A is an $m \times n$ matrix (the constraint matrix),
- $\mathbf{b}$ is an m-dimensional vector (the right-hand side),
- $\gtreqless$ is an m-dimensional vector of relations from $\{\leq, =, \geq\}$,
- $\mathbf{l}$ is an n-dimensional vector of lower bounds for $\mathbf{x}$, where $l_j \in \mathbb{R} \cup \{-\infty\}$ for all j
- $\mathbf{u}$ is an n-dimensional vector of upper bounds for $\mathbf{x}$, where $u_j \in \mathbb{R} \cup \{\infty\}$ for all j
- D is a symmetric positive-semidefinite $n \times n$ matrix (the quadratic objective function),
- $\mathbf{c}$ is an n-dimensional vector (the linear objective function), and
- c_0 is a constant.

If $D = 0$, the program is a linear program; if the variable bounds are $x \geq 0$, we have a nonnegative program.

The program data are read from an input stream in *MPSFormat*. This is a commonly used format for encoding linear and quadratic programs that is understood by many solvers. All values are expected to be readable into type *NT*. The constructed program can be further manipulated by using the set-methods below.

Is Model for the Concepts

QuadraticProgram
LinearProgram
NonnegativeQuadraticProgram
NonnegativeLinearProgram

Types

Quadratic_program_from_mps<NT>:: NT	The number type of the program entries.

Creation

Quadratic_program_from_mps<NT> qp(std::istream& in);

reads *qp* from the input stream *in*.

Operations

bool *qp.is_valid() const*
returns *true* if and only if an MPS-encoded quadratic program could be extracted from the input stream.

std::string *qp.get_error() const*
if !*qp.is_valid()*, this method returns an error message explaining why the input does not conform to the *MPSFormat*.

std::string *qp.variable_name_by_index(int j) const*

returns the name of the j-th variable.
Precondition: j must not refer to a variable that has been added later, using one of the set methods below.

int *qp.variable_index_by_name(std::string name) const*

returns the index of the variable with name *name*. If there is no variable with this name, the result is -1.

std::string *qp.constraint_name_by_index(int i) const*

returns the name of the i-th constraint.
Precondition: i must not refer to a constraint that has been added later, using one of the set methods below.

int *qp.constraint_index_by_name(std::string name) const*

returns the index of the constraint with name *name*. If there is no constraint with this name, the result is -1.

bool *qp.is_linear() const*
returns *true* if and only if *qp* is a linear program.

bool *qp.is_nonnegative() const*

returns *true* if and only if *qp* is a nonnegative program.

void *qp.set_a(int j, int i, NT val)*

sets the entry A_{ij} in column j and row i of the constraint matrix A of *qp* to *val*. An existing entry is overwritten. *qp* is enlarged if necessary to accomodate this entry.

void *qp.set_b(int i, NT val)*

sets the entry b_i of *qp* to *val*. An existing entry is overwritten. *qp* is enlarged if necessary to accomodate this entry.

void *qp.set_r(int i, CGAL::Comparison_result rel)*

sets the entry $\gtreqless_i$ of *qp* to *rel*. *CGAL::SMALLER* means that the *i*-th constraint is of type "$\leq$", *CGAL::EQUAL* means "$=$", and *CGAL::LARGER* encodes "$\geq$". An existing entry is overwritten. *qp* is enlarged if necessary to accomodate this entry.

void *qp.set_l(int j, bool is_finite, NT val = NT(0))*

if *is_finite*, this sets the entry l_j of *qp* to *val*, otherwise it sets l_j to $-\infty$. An existing entry is overwritten. *qp* is enlarged if necessary to accomodate this entry.

void *qp.set_u(int j, bool is_finite, NT val = NT(0))*

if *is_finite*, this sets the entry u_j of *qp* to *val*, otherwise it sets u_j to ∞. An existing entry is overwritten. *qp* is enlarged if necessary to accomodate this entry.

void *qp.set_c(int j, NT val)*

sets the entry c_j of *qp* to *val*. An existing entry is overwritten. *qp* is enlarged if necessary to accomodate this entry.

void *qp.set_c0(NT val)* sets the entry c_0 of *qp* to *val*. An existing entry is overwritten.

void *qp.set_d(int i, int j, NT val)*

sets the entries $2D_{ij}$ and $2D_{ji}$ of *qp* to *val*. Existing entries are overwritten. *qp* is enlarged if necessary to accomodate these entries.
Precondition: *j <= i*

Example

QP_solver/first_qp_from_mps.cpp
QP_solver/first_lp_from_mps.cpp
QP_solver/first_nonnegative_qp_from_mps.cpp
QP_solver/first_nonnegative_lp_from_mps.cpp

See Also

Quadratic_program<NT>
Quadratic_program_from_iterators<A_it, B_it, R_it, FL_it, L_it, FU_it, U_it, D_it, C_it>
Linear_program_from_iterators<A_it, B_it, R_it, FL_it, L_it, FU_it, U_it, C_it>
Nonnegative_quadratic_program_from_iterators<A_it, B_it, R_it, D_it, C_it>
Nonnegative_linear_program_from_iterators<A_it, B_it, R_it, C_it>

Class

CGAL::Quadratic_program_from_iterators<A_it, B_it, R_it, FL_it, L_it, FU_it, U_it, D_it, C_it>

#include <CGAL/QP_models.h>

Definition

An object of class *Quadratic_program_from_iterators<A_it, B_it, R_it, FL_it, L_it, FU_it, U_it, D_it, C_it>* describes a convex quadratic program of the form

$$
\begin{array}{lll}
\text{(QP)} & \text{minimize} & \mathbf{x}^T D\mathbf{x} + \mathbf{c}^T\mathbf{x} + c_0 \\
 & \text{subject to} & A\mathbf{x} \gtreqless \mathbf{b}, \\
 & & \mathbf{l} \le \mathbf{x} \le \mathbf{u}
\end{array}
$$

in n real variables $\mathbf{x} = (x_0,\ldots,x_{n-1})$. Here,

- A is an $m \times n$ matrix (the constraint matrix),
- $\mathbf{b}$ is an m-dimensional vector (the right-hand side),
- $\gtreqless$ is an m-dimensional vector of relations from $\{\le, =, \ge\}$,
- $\mathbf{l}$ is an n-dimensional vector of lower bounds for $\mathbf{x}$, where $l_j \in \mathbb{R} \cup \{-\infty\}$ for all j
- $\mathbf{u}$ is an n-dimensional vector of upper bounds for $\mathbf{x}$, where $u_j \in \mathbb{R} \cup \{\infty\}$ for all j
- D is a symmetric positive-semidefinite $n \times n$ matrix (the quadratic objective function),
- $\mathbf{c}$ is an n-dimensional vector (the linear objective function), and
- c_0 is a constant.

This class is simply a wrapper for existing iterators, and it does not copy the program data.

It frequently happens that all values in one of the vectors from above are the same, for example if the system $Ax \gtreqless b$ is actually a system of equations $Ax = b$. To get an iterator over such a vector, it is not necessary to store multiple copies of the value in some container; an instance of the class *Const_oneset_iterator<T>*, constructed from the value in question, does the job more efficiently.

Is Model for the Concepts

QuadraticProgram

Creation

Quadratic_program_from_iterators<A_it, B_it, R_it, FL_it, L_it, FU_it, U_it, D_it, C_it> qp(int n,
int m,
A_it a,
B_it b,
R_it r,
FL_it fl,
L_it l,
FU_it fu,
U_it u,
D_it d,
C_it c,
std::iterator_traits<C_it>value_type c0 = 0)

constructs *qp* from given random-access iterators and the constant *c0*. The passed iterators are merely stored, no copying of the program data takes place. How these iterators are supposed to encode the quadratic program is described in *QuadraticProgram*.

Example

QP_solver/first_qp_from_iterators.cpp

The following example for the simpler model *Nonnegative_linear_program_from_iterators<A_it, B_it, R_it, C_it>* should give you a flavor of the use of this model in practice.

QP_solver/solve_convex_hull_containment_lp.h
QP_solver/convex_hull_containment.cpp **See Also**

QuadraticProgram Quadratic_program<NT>
Quadratic_program_from_mps<NT>

Class

CGAL::Linear_program_from_iterators<A_it, B_it, R_it, FL_it, L_it, FU_it, U_it, C_it>

#include <CGAL/QP_models.h>

Definition

An object of class *Linear_program_from_iterators<A_it, B_it, R_it, FL_it, L_it, FU_it, U_it, C_it>* describes a linear program of the form

$$\begin{array}{lll} \text{(QP)} & \text{minimize} & \mathbf{c}^T\mathbf{x}+c_0 \\ & \text{subject to} & A\mathbf{x} \gtreqless \mathbf{b}, \\ & & \mathbf{l} \leq \mathbf{x} \leq \mathbf{u} \end{array}$$

in n real variables $\mathbf{x} = (x_0, \ldots, x_{n-1})$. Here,

- A is an $m \times n$ matrix (the constraint matrix),
- $\mathbf{b}$ is an m-dimensional vector (the right-hand side),
- $\gtreqless$ is an m-dimensional vector of relations from $\{\leq, =, \geq\}$,
- $\mathbf{l}$ is an n-dimensional vector of lower bounds for $\mathbf{x}$, where $l_j \in \mathbb{R} \cup \{-\infty\}$ for all j
- $\mathbf{u}$ is an n-dimensional vector of upper bounds for $\mathbf{x}$, where $u_j \in \mathbb{R} \cup \{\infty\}$ for all j
- $\mathbf{c}$ is an n-dimensional vector (the linear objective function), and
- c_0 is a constant.

This class is simply a wrapper for existing iterators, and it does not copy the program data.

It frequently happens that all values in one of the vectors from above are the same, for example if the system $Ax \gtreqless b$ is actually a system of equations $Ax = b$. To get an iterator over such a vector, it is not necessary to store multiple copies of the value in some container; an instance of the class *Const_oneset_iterator<T>*, constructed from the value in question, does the job more efficiently.

Is Model for the Concepts

QuadraticProgram
LinearProgram

Creation

Linear_program_from_iterators<A_it, B_it, R_it, FL_it, L_it, FU_it, U_it, C_it> lp(int n,
int m,
A_it a,
B_it b,
R_it r,
FL_it fl,
L_it l,
FU_it fu,
U_it u,
C_it c,
std::iterator_traits<C_it>
value_type c0 = 0)

constructs *lp* from given random-access iterators and the constant *c0*. The passed iterators are merely stored, no copying of the program data takes place. How these iterators are supposed to encode the linear program is described in *LinearProgram*.

Example

QP_solver/first_lp_from_iterators.cpp

The following example for the simpler model *Nonnegative_linear_program_from_iterators<A_it, B_it, R_it, C_it>* should give you a flavor of the use of this model in practice.

QP_solver/solve_convex_hull_containment_lp.h
QP_solver/convex_hull_containment.cpp

See Also

LinearProgram Quadratic_program<NT>
Quadratic_program_from_mps<NT>

CGAL::Nonnegative_quadratic_program_from_iterators<A_it, B_it, R_it, D_it, C_it>

#include <CGAL/QP_models.h>

Definition

An object of class *Nonnegative_quadratic_program_from_iterators<A_it, B_it, R_it, D_it, C_it>* describes a convex quadratic program of the form

$$
\begin{array}{lll}
\text{(QP)} & \text{minimize} & \mathbf{x}^T D\mathbf{x} + \mathbf{c}^T\mathbf{x} + c_0 \\
 & \text{subject to} & A\mathbf{x} \gtreqless \mathbf{b}, \\
 & & \mathbf{x} \geq 0
\end{array}
$$

in n real variables $\mathbf{x} = (x_0, \ldots, x_{n-1})$. Here,

- A is an $m \times n$ matrix (the constraint matrix),
- $\mathbf{b}$ is an m-dimensional vector (the right-hand side),
- $\gtreqless$ is an m-dimensional vector of relations from $\{\leq, =, \geq\}$,
- D is a symmetric positive-semidefinite $n \times n$ matrix (the quadratic objective function),
- $\mathbf{c}$ is an n-dimensional vector (the linear objective function), and
- c_0 is a constant.

This class is simply a wrapper for existing iterators, and it does not copy the program data.

It frequently happens that all values in one of the vectors from above are the same, for example if the system $Ax \gtreqless b$ is actually a system of equations $Ax = b$. To get an iterator over such a vector, it is not necessary to store multiple copies of the value in some container; an instance of the class *Const_oneset_iterator<T>*, constructed from the value in question, does the job more efficiently.

Is Model for the Concepts

QuadraticProgram
NonnegativeQuadraticProgram

Creation

Nonnegative_quadratic_program_from_iterators<A_it, B_it, R_it, D_it, C_it> qp(int n,
int m,
A_it a,
B_it b,
R_it r,
D_it d,
C_it c,
std::iterator_traits<C_it>

value_type c0 = 0)

constructs *qp* from given random-access iterators and the constant *c0*. The passed iterators are merely stored, no copying of the program data takes place. How these iterators are supposed to encode the nonnegative quadratic program is described in *NonnegativeQuadraticProgram*.

Example

QP_solver/first_nonnegative_qp_from_iterators.cpp

The following example for the simpler model *Nonnegative_linear_program_from_iterators<A_it, B_it, R_it, C_it>* should give you a flavor of the use of this model in practice.

QP_solver/solve_convex_hull_containment_lp.h
QP_solver/convex_hull_containment.cpp

See Also

NonnegativeQuadraticProgram *Quadratic_program<NT>*
Quadratic_program_from_mps<NT>

Class

CGAL::Nonnegative_linear_program_from_iterators<A_it, B_it, R_it, C_it>

#include <CGAL/QP_models.h>

Definition

An object of class *Nonnegative_linear_program_from_iterators<A_it, B_it, R_it, C_it>* describes a linear program of the form

$$\begin{array}{lll} \text{(QP)} & \text{minimize} & \mathbf{c}^T\mathbf{x} + c_0 \\ & \text{subject to} & A\mathbf{x} \gtreqless \mathbf{b}, \\ & & \mathbf{x} \geq 0 \end{array}$$

in n real variables $\mathbf{x} = (x_0, \ldots, x_{n-1})$. Here,

- A is an $m \times n$ matrix (the constraint matrix),
- $\mathbf{b}$ is an m-dimensional vector (the right-hand side),
- $\gtreqless$ is an m-dimensional vector of relations from $\{\leq, =, \geq\}$,
- $\mathbf{c}$ is an n-dimensional vector (the linear objective function), and
- c_0 is a constant.

This class is simply a wrapper for existing iterators, and it does not copy the program data.

It frequently happens that all values in one of the vectors from above are the same, for example if the system $Ax \gtreqless b$ is actually a system of equations $Ax = b$. To get an iterator over such a vector, it is not necessary to store multiple copies of the value in some container; an instance of the class *Const_oneset_iterator<T>*, constructed from the value in question, does the job more efficiently.

Is Model for the Concepts

QuadraticProgram
LinearProgram
NonnegativeQuadraticProgram
NonnegativeLinearProgram

Creation

Nonnegative_linear_program_from_iterators<A_it, B_it, R_it, C_it> lp(int n,
int m,
A_it a,
B_it b,
R_it r,
C_it c,
std::iterator_traits<C_it>value_type c0

= 0)

constructs *lp* from given random-access iterators and the constant *c0*. The passed iterators are merely stored, no copying of the program data takes place. How these iterators are supposed to encode the nonnegative linear program is described in *NonnegativeLinearProgram*.

Example

QP_solver/first_nonnegative_lp_from_iterators.cpp
QP_solver/solve_convex_hull_containment_lp.h
QP_solver/convex_hull_containment.cpp

See Also

NonnegativeLinearProgram Quadratic_program<NT>
Quadratic_program_from_mps<NT>

Function

CGAL::make_quadratic_program_from_iterators

#include <CGAL/QP_models.h>

This template function creates an instance of *Quadratic_program_from_iterators<A_it, B_it, R_it, FL_it, L_it, FU_it, U_it, D_it, C_it>* from given iterators. This function can be useful if the types of these iterators are too complicated (or of too little interest for you) to write them down explicitly.

template < A_it, B_it, R_it, FL_it, L_it, FU_it, U_it, D_it, C_it >
Quadratic_program_from_iterators<A_it, B_it, R_it, FL_it, L_it, FU_it, U_it, D_it, C_it>

make_quadratic_program_from_iterators(int n,
int m,
A_it a,
B_it b,
R_it r,
FL_it fl,
L_it l,
FU_it fu,
U_it u,
D_it d,
C_it c,
std::iterator_traits<C_it>::value_type c0 = std::iterator_traits<C_it>::value_type(0))

returns an instance of *Quadratic_program_from_iterators< A_it, B_it, R_it, FL_it, L_it, FU_it, U_it, D_it, C_it>*, constructed from the given iterators.

Example

The following example demonstrates the typical usage of makers with the simpler function *make_nonnegative_linear_program_from_iterators*.

QP_solver/solve_convex_hull_containment_lp2.h
QP_solver/convex_hull_containment2.cpp

See Also

Quadratic_program_from_iterators<A_it, B_it, R_it, FL_it, L_it, FU_it, U_it, D_it, C_it>

CGAL::make_linear_program_from_iterators

#include <CGAL/QP_models.h>

This template function creates an instance of *Linear_program_from_iterators<A_it, B_it, R_it, FL_it, L_it, FU_it, U_it, C_it>* from given iterators. This function can be useful if the types of these iterators are too complicated (or of too little interest for you) to write them down explicitly.

template < A_it, B_it, R_it, FL_it, L_it, FU_it, U_it, C_it >
Linear_program_from_iterators<A_it, B_it, R_it, FL_it, L_it, FU_it, U_it, C_it>

make_linear_program_from_iterators(int n,
int m,
A_it a,
B_it b,
R_it r,
FL_it fl,
L_it l,
FU_it fu,
U_it u,
C_it c,
std::iterator_traits<C_it>::value_type c0 = std::iterator_traits<C_it>::value_type(0))

returns an instance of *Linear_program_from_iterators<A_it, B_it, R_it, FL_it, L_it, FU_it, U_it, C_it>*, constructed from the given iterators.

Example

The following example demonstrates the typical usage of makers with the simpler function *make_nonnegative_linear_program_from_iterators*.

QP_solver/solve_convex_hull_containment_lp2.h
QP_solver/convex_hull_containment2.cpp

See Also

Linear_program_from_iterators<A_it, B_it, R_it, FL_it, L_it, FU_it, U_it, C_it>

Function

Function

CGAL::make_nonnegative_quadratic_program_from_iterators

#include <CGAL/QP_models.h>

This template function creates an instance of *Nonnegative_quadratic_program_from_iterators<A_it, B_it, R_it, D_it, C_it>* from given iterators. This function can be useful if the types of these iterators are too complicated (or of too little interest for you) to write them down explicitly.

template < A_it, B_it, R_it, D_it, C_it >
Nonnegative_quadratic_program_from_iterators <A_it, B_it, R_it, D_it, C_it>

make_nonnegative_quadratic_program_from_iterators(int n,
int m,
A_it a,
B_it b,
R_it r,
D_it d,
C_it c,
std::iterator_traits<C_it>
::value_type c0 = std::iterator_traits<C_it>::value_type(0))

returns an instance of *Nonnegative_quadratic_program_from_iterators<A_it, B_it, R_it, D_it, C_it>*, constructed from the given iterators.

Example

The following example demonstrates the typical usage of makers with the simpler function *make_nonnegative_linear_program_from_iterators*.

QP_solver/solve_convex_hull_containment_lp2.h
QP_solver/convex_hull_containment2.cpp

See Also

Nonnegative_quadratic_program_from_iterators<A_it, B_it, R_it, D_it, C_it>

CGAL::make_nonnegative_linear_program_from_iterators

#include <CGAL/QP_models.h>

This template function creates an instance of *Nonnegative_linear_program_from_iterators<A_it, B_it, R_it, C_it>* from given iterators. This function can be useful if the types of these iterators are too complicated (or of too little interest for you) to write them down explicitly.

template < A_it, B_it, R_it, C_it >
Nonnegative_linear_program_from_iterators <A_it, B_it, R_it, C_it>

make_nonnegative_linear_program_from_iterators(int n, int m, A_it a, B_it b, R_it r, C_it c, std::iterator_traits<C_it>::value_type c0 = std::iterator_traits<C_it>::value_type(0))

returns an instance of *Nonnegative_linear_program_from_iterators<A_it, B_it, R_it, C_it>*, constructed from the given iterators.

Example

QP_solver/solve_convex_hull_containment_lp2.h
QP_solver/convex_hull_containment2.cpp

See Also

Nonnegative_linear_program_from_iterators<A_it, B_it, R_it, C_it>

Function

CGAL::solve_quadratic_program

#include <CGAL/QP_functions.h>

This function solves a quadratic program, using some exact Integral Domain *ET* for its computations. Various options may be provided, see *Quadratic_program_options*.

template <QuadraticProgram, ET>
Quadratic_program_solution<ET>

solve_quadratic_program(QuadraticProgram qp,
ET,
Quadratic_program_options options = Quadratic_program_options())

returns the solution of the quadratic program *qp*, solved with exact number type *ET*.

Requirements

ET is a model of the concepts *IntegralDomain* and *RealEmbeddable*; it must be an exact type, and all entries of *qp* are convertible to *ET*.

Here are some recommended combinations of input type (the type of the *qp* entries) and *ET*.

input type	—	*ET*
double	—	*MP_Float*, *Gmpzf*, or *Gmpq*
int	—	*MP_Float*, or *Gmpz*
any exact type *NT*	—	*NT*

Note: by default, this function performs a large number of runtime-checks to ensure consistency during the solution process. However, these checks slow down the computations by a considerable factor. For maximum efficiency, it is advisable to define the macros `CGAL_QP_NO_ASSERTIONS` or `NDEBUG`.

Example

QP_solver/first_qp.cpp

See Also

The models of *QuadraticProgram* .... page 500:

Quadratic_program<NT>
Quadratic_program_from_mps<NT>
Quadratic_program_from_iterators<A_it, B_it, R_it, FL_it, L_it, FU_it, U_it, D_it, C_it>

CGAL::solve_linear_program

#include <CGAL/QP_functions.h>

This function solves a linear program, using some exact Integral Domain *ET* for its computations. Various options may be provided, see *Quadratic_program_options*.

template <LinearProgram, ET>
Quadratic_program_solution<ET>

solve_linear_program(LinearProgram lp,
ET,
Quadratic_program_options options = Quadratic_program_options())

returns the solution of the linear program *lp*, solved with exact number type *ET*.

Requirements

ET is a model of the concepts *IntegralDomain* and *RealEmbeddable*; it must be an exact type, and all entries of *qp* are convertible to *ET*.

Here are some recommended combinations of input type (the type of the *qp* entries) and *ET*.

input type	—	*ET*
double	—	*MP_Float*, *Gmpzf*, or *Gmpq*
int	—	*MP_Float*, or *Gmpz*
any exact type *NT*	—	*NT*

Note: by default, this function performs a large number of runtime-checks to ensure consistency during the solution process. However, these checks slow down the computations by a considerable factor. For maximum efficiency, it is advisable to define the macros `CGAL_QP_NO_ASSERTIONS` or `NDEBUG`.

Example

QP_solver/first_lp.cpp

See Also

The models of *LinearProgram* . . . . page 506:

Quadratic_program<NT>
Quadratic_program_from_mps<NT>
Linear_program_from_iterators<A_it, B_it, R_it, FL_it, L_it, FU_it, U_it, C_it>

Function

CGAL::solve_nonnegative_quadratic_program

#include <CGAL/QP_functions.h>

This function solves a nonnegative quadratic program, using some exact Integral Domain *ET* for its computations. Various options may be provided, see *Quadratic_program_options*.

template <NonnegativeQuadraticProgram, ET>
Quadratic_program_solution<ET>

solve_nonnegative_quadratic_program(NonnegativeQuadraticProgram qp, ET, Quadratic_program_options options = Quadratic_program_options())

returns the solution of the nonnegative quadratic program *qp*, solved with exact number type *ET*.

Requirements

ET is a model of the concepts *IntegralDomain* and *RealEmbeddable*; it must be an exact type, and all entries of *qp* are convertible to *ET*.

Here are some recommended combinations of input type (the type of the *qp* entries) and *ET*.

input type	—	*ET*
double	—	*MP_Float*, *Gmpzf*, or *Gmpq*
int	—	*MP_Float*, or *Gmpz*
any exact type *NT*	—	*NT*

Note: by default, this function performs a large number of runtime-checks to ensure consistency during the solution process. However, these checks slow down the computations by a considerable factor. For maximum efficiency, it is advisable to define the macros `CGAL_QP_NO_ASSERTIONS` or `NDEBUG`.

Example

QP_solver/first_nonnegative_qp.cpp

See Also

The models of *NonnegativeQuadraticProgram* . . . . page 503:

Quadratic_program<NT>
Quadratic_program_from_mps<NT>
Nonnegative_quadratic_program_from_iterators<A_it, B_it, R_it, D_it, C_it>

CGAL::solve_nonnegative_linear_program

#include <CGAL/QP_functions.h>

This function solves a nonnegative linear program, using some exact Integral Domain *ET* for its computations. Various options may be provided, see *Quadratic_program_options*.

template <NonnegativeLinearProgram, ET>
Quadratic_program_solution<ET>

solve_nonnegative_linear_program(NonnegativeLinearProgram lp, ET, Quadratic_program_options options = Quadratic_program_options())

returns the solution of the nonnegative linear program *lp*, solved with exact number type *ET*.

Requirements

ET is a model of the concepts *IntegralDomain* and *RealEmbeddable*; it must be an exact type, and all entries of *qp* are convertible to *ET*.

Here are some recommended combinations of input type (the type of the *qp* entries) and *ET*.

input type	—	*ET*
double	—	*MP_Float*, *Gmpzf*, or *Gmpq*
int	—	*MP_Float*, or *Gmpz*
any exact type *NT*	—	*NT*

Note: by default, this function performs a large number of runtime-checks to ensure consistency during the solution process. However, these checks slow down the computations by a considerable factor. For maximum efficiency, it is advisable to define the macros `CGAL_QP_NO_ASSERTIONS` or `NDEBUG`.

Example

QP_solver/first_nonnegative_lp.cpp

See Also

The models of *NonnegativeLinearProgram* . . . . page 509:

Quadratic_program<NT>
Quadratic_program_from_mps<NT>
Nonnegative_linear_program_from_iterators<A_it, B_it, R_it, C_it>

Class

CGAL::Quadratic_program_options

#include <CGAL/QP_options.h>

Definition

This is a class used for passing options to the linear and quadratic programming solvers. Currently, we support only options referring to

1. the verbosity,
2. the pricing strategy (see *Quadratic_program_pricing_strategy*),
3. the validation mode (see the Validity section of *Quadratic_program_solution*)

The idea is that this list grows in the future.

Creation

Quadratic_program_options options; — constructs an instance of *Quadratic_program_options* where all available options are at their defaults.

Operations

Here we just have set/get pairs for any option type.

Verbosity

void — *options.set_verbosity(int verbosity)*

sets the verbosity of the solver to the value *verbosity* when *options* is passed to any of the four solution functions. The provided value must be a number between 0 and 5. Verbosity 0 is the default and results in the solver running silently. Verbosity 1 prints a short summary of every iteration. Higher verbosity values print more information about the solution process, but these are mainly for debugging purposes and have no effect if you compile with `CGAL_QP_NO_ASSERTIONS` or `NDEBUG`.

int — *options.get_verbosity() const*

returns the verbosity level of *options*.

Pricing strategy

void *options.set_pricing_strategy(Quadratic_program_pricing_strategy pricing_strategy)*

sets the pricing strategy of the solver to the value *pricing_strategy* when *options* is passed to any of the four solution functions. The pricing strategy controls how the solver proceeds from any intermediate solution. For the available strategies and their behavior, see the documentation of the class *Quadratic_program_pricing_strategy*.

Quadratic_program_pricing_strategy

options.get_pricing_strategy() const

returns the pricing strategy of *options*.

Validation mode

void *options.set_auto_validation(bool validate)*

sets the automatic validation mode of the solver to the value *validate*. The default is *false*. By providing value *true* you can tell the solver to automatically check whether the program has correctly been solved, see the Validity section of the class *Quadratic_program_solution*.

bool *options.get_auto_validation() const*

returns the validation mode of *options*.

Example

QP_solver/cycling.cpp

See Also

Quadratic_program_solution
solve_quadratic_program
solve_linear_program
solve_nonnegative_quadratic_program
solve_nonnegative_linear_program

Class

CGAL::Quadratic_program_pricing_strategy

#include <CGAL/QP_options.h>

Definition

This is an enumeration type containing the values *QP_CHOOSE_DEFAULT*, *QP_DANTZIG*, *QP_PARTIAL_DANTZIG*, *QP_FILTERED_DANTZIG*, *QP_PARTIAL_FILTERED_DANTZIG*, and*QP_BLAND*.

It indicates the pricing strategy to be used in solving a linear or quadratic program. This strategy determines how the solver gets from one intermediate solution to the next during any of its iterations.

Here we briefly describe when to choose which strategy.

QP_CHOOSE_DEFAULT

This is the default value of the pricing strategy in *Quadratic_program_options*, and it lets the solver choose the strategy that it thinks is most appropriate for the problem at hand. There are only few reasons to deviate from this default, but you are free to experiment, of course.

QP_PARTIAL_DANTZIG

If the input type is **not** *double*, this is usually the best choice for linear and quadratic programs of medium size.

QP_DANTZIG

If the input type is **not** *double*, this can sometimes make a difference (be faster or slowe) than *QP_PARTIAL_DANTZIG* for problems with a high variable/constraint or constraint/variable ratio.

QP_PARTIAL_FILTERED_DANTZIG

If the input type **is** *double*, this is usually the best choice for linear and quadratic programs of medium size. If the input type is not *double*, this choice is equivalent to *QP_PARTIAL_DANTZIG*.

Note: filtered strategies may in rare cases fail due to double exponent overflows, see Section 10.8.1. In this case, the slower fallback option is the non-filtered variant *QP_PARTIAL_DANTZIG* of this strategy.

QP_FILTERED_DANTZIG

If the input type **is** *double*, this can sometimes make a difference (be faster or slowe) than *QP_PARTIAL_FILTERED_DANTZIG* for problems with a high variable/constraint or constraint/variable ratio. If the input type is not *double*, this choice is equivalent to *QP_DANTZIG*.

Note: filtered strategies may in rare cases fail due to double exponent overflows, see Section 10.8.1. In this case, the slower fallback option is the non-filtered variant *QP_DANTZIG* of this strategy.

QP_BLAND

This is hardly ever the most efficient choice, but it is guaranteed to avoid internal cycling of the solution algorithm, see Section 10.8.2.

See Also

Quadratic_program_options

CGAL::print_quadratic_program

#include <CGAL/QP_functions.h>

This function writes a quadratic program to an output stream (in *MPSFormat*). The time complexity is $\Theta(n^2 + mn)$, even if the program is very sparse.

template <QuadraticProgram>
void *print_quadratic_program(std::ostream& out, QuadraticProgram qp, std::string problem_name = std::string("MY_MPS"))*

writes the quadratic program *qp* to *out* in *MPSFormat*. The name of the program will be the one provided by *problem_name*.

Requirements

Output operators are defined for all entry types of *qp*.

Example

QP_solver/print_first_qp.cpp

See Also

The concept

QuadraticProgram

Function

CGAL::print_linear_program

#include <CGAL/QP_functions.h>

This function writes a linear program to an output stream (in *MPSFormat*). The time complexity is $\Theta(mn)$, even if the program is very sparse.

template <LinearProgram>
void *print_linear_program(std::ostream& out, LinearProgram lp, std::string problem_name = std::string("MY_MPS"))*

writes the linear program *lp* to *out* in *MPSFormat*. The name of the program will be the one provided by *problem_name*.

Requirements

Output operators are defined for all entry types of *lp*.

Example

QP_solver/print_first_lp.cpp

See Also

The concept

LinearProgram

Function

CGAL::print_nonnegative_quadratic_program

#include <CGAL/QP_functions.h>

This function writes a nonnegative quadratic program to an output stream (in *MPSFormat*). The time complexity is $\Theta(n^2 + mn)$, even if the program is very sparse.

template <NonnegativeQuadraticProgram>
void *print_nonnegative_quadratic_program(std::ostream& out, NonnegativeQuadraticProgram qp, std::string problem_name = std::string("MY_MPS"))*

writes the nonnegative quadratic program *qp* to *out* in *MPSFormat*. The name of the program will be the one provided by *problem_name*.

Requirements

Output operators are defined for all entry types of *qp*.

Example

QP_solver/print_first_nonnegative_qp.cpp

See Also

The concept

NonnegativeQuadraticProgram

CGAL::print_nonnegative_linear_program

#include <CGAL/QP_functions.h>

This function writes a nonnegative linear program to an output stream (in *MPSFormat*). The time complexity is $\Theta(mn)$, even if the program is very sparse.

template <NonnegativeLinearProgram>
void print_nonnegative_linear_program(std::ostream& out, NonnegativeLinearProgram lp, std::string problem_name = std::string("MY_MPS"))

writes the nonnegative linear program *lp* to *out* in *MPSFormat*. The name of the program will be the one provided by *problem_name*.

Requirements

Output operators are defined for all entry types of *lp*.

Example

QP_solver/print_first_nonnegative_lp.cpp

See Also

The concept

NonnegativeLinearProgram

Function

Part IV

Geometry Kernels

Chapter 11

2D and 3D Geometry Kernel

Hervé Brönnimann, Andreas Fabri, Geert-Jan Giezeman, Susan Hert, Michael Hoffmann, Lutz Kettner, Sylvain Pion, and Stefan Schirra

Contents

11.1 Introduction

CGAL, the *Computational Geometry Algorithms Library*, is written in C++ and consists of three major parts. The first part is the kernel, which consists of constant-size non-modifiable geometric primitive objects and operations on these objects. The objects are represented both as stand-alone classes that are parameterized by a representation class, which specifies the underlying number types used for calculations and as members of the kernel classes, which allows for more flexibility and adaptability of the kernel. The second part is a collection of basic geometric data structures and algorithms, which are parameterized by traits classes that define the interface between the data structure or algorithm and the primitives they use. In many cases, the kernel classes provided in CGAL can be used as traits classes for these data structures and algorithms. The third part of the library consists of non-geometric support facilities, such as circulators, random sources, I/O support for debugging and for interfacing CGAL to various visualization tools.

This part of the reference manual covers the kernel. The kernel contains objects of constant size, such as point, vector, direction, line, ray, segment, triangle, iso-oriented rectangle and tetrahedron. With each type comes a set of functions which can be applied to an object of this type. You will typically find access functions (e.g. to the coordinates of a point), tests of the position of a point relative to the object, a function returning the bounding box, the length, or the area of an object, and so on. The CGAL kernel further contains basic operations such as affine transformations, detection and computation of intersections, and distance computations.

11.1.1 Robustness

The correctness proof of nearly all geometric algorithms presented in theory papers assumes exact computation with real numbers. This leads to a fundamental problem with the implementation of geometric algorithms. Naively, often the exact real arithmetic is replaced by inexact floating-point arithmetic in the implementation. This often leads to acceptable results for many input data. However, even for the implementation of the simplest geometric algorithms this simplification occasionally does not work. Rounding errors introduced by an inaccurate arithmetic may lead to inconsistent decisions, causing unexpected failures for some correct input data. There are many approaches to this problem, one of them is to compute exactly (compute so accurate that all decisions made by the algorithm are exact) which is possible in many cases but more expensive than standard floating-point arithmetic. C. M. Hoffmann [Hof89a, Hof89b] illustrates some of the problems arising in the implementation of geometric algorithms and discusses some approaches to solve them. A more recent overview is given in [Sch00]. The exact computation paradigm is discussed by Yap and Dubé [YD95] and Yap [Yap97].

In CGAL you can choose the underlying number types and arithmetic. You can use different types of arithmetic simultaneously and the choice can be easily changed, e.g. for testing. So you can choose between implementations with fast but occasionally inexact arithmetic and implementations guaranteeing exact computation and exact results. Of course you have to pay for the exactness in terms of execution time and storage space. See the dedicated chapter for more details on number types and their capabilities and performance.

11.2 Kernel Representations

Our object of study is the d-dimensional affine Euclidean space. Here we are mainly concerned with cases $d = 2$ and $d = 3$. Objects in that space are sets of points. A common way to represent the points is the use of Cartesian coordinates, which assumes a reference frame (an origin and d orthogonal axes). In that framework, a point is represented by a d-tuple $(c_0, c_1, \dots, c_{d-1})$, and so are vectors in the underlying linear space. Each point is represented uniquely by such Cartesian coordinates. Another way to represent points is by homogeneous coordinates. In that framework, a point is represented by a $(d+1)$-tuple $(h_0, h_1, \dots, h_d)$. Via the formulae $c_i = h_i/h_d$, the corresponding point with Cartesian coordinates $(c_0, c_1, \dots, c_{d-1})$ can be computed. Note that homogeneous coordinates are not unique. For $\lambda \neq 0$, the tuples $(h_0, h_1, \dots, h_d)$ and $(\lambda \cdot h_0, \lambda \cdot h_1, \dots, \lambda \cdot h_d)$ represent the same point. For a point with Cartesian coordinates $(c_0, c_1, \dots, c_{d-1})$ a possible homogeneous representation is $(c_0, c_1, \dots, c_{d-1}, 1)$. Homogeneous coordinates in fact allow to represent objects in a more general space, the projective space $\mathbb{P}^d$. In CGAL, we do not compute in projective geometry. Rather, we use homogeneous coordinates to avoid division operations, since the additional coordinate can serve as a common denominator.

11.2.1 Genericity Through Parameterization

Almost all the kernel objects (and the corresponding functions) are templates with a parameter that allows the user to choose the representation of the kernel objects. A type that is used as an argument for this parameter must fulfill certain requirements on syntax and semantics. The list of requirements defines an abstract kernel concept. For all kernel objects types, the types *CGAL::Type<Kernel>* and *Kernel::Type* are identical.

CGAL offers four families of concrete models for the concept Kernel, two based on the Cartesian representation of points and two based on the homogeneous representation of points. The interface of the kernel objects is designed such that it works well with both Cartesian and homogeneous representation. For example, points in 2D have a constructor with three arguments as well (the three homogeneous coordinates of the point). The common interfaces parameterized with a kernel class allow one to develop code independent of the chosen representation. We said "families" of models, because both families are parameterized too. A user can choose the number type used to represent the coordinates.

For reasons that will become evident later, a kernel class provides two typenames for number types, namely *Kernel::FT* and *Kernel::RT*. The type *Kernel::FT* must fulfill the requirements on what is called a *FieldNumberType* in CGAL. This roughly means that *Kernel::FT* is a type for which operations $+$, $-$, $*$ and $/$ are defined with semantics (approximately) corresponding to those of a field in a mathematical sense. Note that, strictly speaking, the built-in type *int* does not fulfill the requirements on a field type, since *int*s correspond to elements of a ring rather than a field, especially operation $/$ is not the inverse of $*$. The requirements on the type *Kernel::RT* are weaker. This type must fulfill the requirements on what is called a *RingNumberType* in CGAL. This roughly means that *Kernel::RT* is a type for which operations $+$, $-$, $*$ are defined with semantics (approximately) corresponding to those of a ring in a mathematical sense.

11.2.2 Cartesian Kernels

With *Cartesian<FieldNumberType>* you can choose a Cartesian representation of coordinates. When you choose Cartesian representation you have to declare at the same time the type of the coordinates. A number type used with the *Cartesian* representation class should be a FieldNumberType as described above. As mentioned above, the built-in type *int* is not a FieldNumberType. However, for some computations with Cartesian representation, no division operation is needed, i.e., a *RingNumberType* is sufficient in this case. With *Cartesian<FieldNumberType>*, both *Cartesian<FieldNumberType>::FT* and *Cartesian<FieldNumberType>::RT* are mapped to *FieldNumberType*.

Cartesian<FieldNumberType> uses reference counting internally to save copying costs. CGAL also provides *Simple_cartesian<FieldNumberType>*, a kernel that uses Cartesian representation but no reference counting. Debugging is easier with *Simple_cartesian<FieldNumberType>*, since the coordinates are stored within the class and hence direct access to the coordinates is possible. Depending on the algorithm, it can also be slightly more or less efficient than *Cartesian<FieldNumberType>*. Again, in *Simple_cartesian<FieldNumberType>* both *Simple_cartesian<FieldNumberType>::FT* and *Simple_cartesian<FieldNumberType>::RT* are mapped to *FieldNumberType*.

11.2.3 Homogeneous Kernels

Homogeneous coordinates permit to avoid division operations in numerical computations, since the additional coordinate can serve as a common denominator. Avoiding divisions can be useful for exact geometric computation. With *Homogeneous<RingNumberType>* you can choose a homogeneous representation for the coordinates of the kernel objects. As for the Cartesian representation, one has to declare the type used to store the coordinates. Since the homogeneous representation does not use divisions, the number type associated with a homogeneous representation class must be a model for the weaker concept *RingNumberType* only. However, some operations provided by this kernel involve divisions, for example computing squared distances or Cartesian coordinates. To keep the requirements on the number type parameter of *Homogeneous* low, the number type *Quotient<RingNumberType>* is used for operations that require divisions. This number type can be viewed as an adaptor which turns a *RingNumberType* into a *FieldNumberType*. It maintains numbers as quotients, i.e., a numerator and a denominator. With *Homogeneous<RingNumberType>*, *Homogeneous<RingNumberType>::FT* is equal to *Quotient<RingNumberType>*, while *Homogeneous<RingNumberType>::RT* is equal to *RingNumberType*.

Homogeneous<RingNumberType> uses reference counting internally to save copying costs. CGAL also provides *Simple_homogeneous<RingNumberType>*, a kernel that uses homogeneous representation but no reference counting. Debugging is easier with *Simple_homogeneous<RingNumberType>*, since the coordinates are stored within the class and hence direct access to the coordinates is possible. Depending on the algorithm, it can also be slightly more or less efficient than *Homogeneous<RingNumberType>*. Again, in *Simple_homogeneous<RingNumberType>* the type *Simple_homogeneous<RingNumberType>::FT* is equal to *Quotient<RingNumberType>* while *Simple_homogeneous<RingNumberType>::RT* is equal to *RingNumberType*.

11.2.4 Naming Conventions

The use of kernel classes not only avoids problems, it also makes all CGAL classes very uniform. They **always** consist of:

1. The *capitalized base name* of the geometric object, such as *Point*, *Segment*, or *Triangle*.
2. An *underscore* followed by the *dimension* of the object, for example _2, _3, or *_d*.
3. A *kernel class* as parameter, which itself is parameterized with a number type, such as *Cartesian<double>* or *Homogeneous<leda_integer>*.

11.2.5 Kernel as a Traits Class

Algorithms and data structures in the basic library of CGAL are parameterized by a traits class that subsumes the objects on which the algorithm or data structure operates as well as the operations to do so. For most of the algorithms and data structures in the basic library you can use a kernel as a traits class. For some algorithms you even do not have to specify the kernel; it is detected automatically using the types of the geometric objects passed to the algorithm. In some other cases, the algorithms or data structures needs more than is provided by the kernel concept. In these cases, a kernel can not be used as a traits class.

11.2.6 Choosing a Kernel and Predefined Kernels

If you start with integral Cartesian coordinates, many geometric computations will involve integral numerical values only. Especially, this is true for geometric computations that evaluate only predicates, which are tantamount to determinant computations. Examples are triangulation of point sets and convex hull computation. In this case, the Cartesian representation is probably the first choice, even with a ring type. You might use limited precision integer types like *int* or *long*, use *double* to present your integers (they have more bits in their mantissa than an *int* and overflow nicely), or an arbitrary precision integer type like the wrapper *Gmpz* for the GMP integers, *leda_integer*, or *MP_Float*. Note, that unless you use an arbitrary precision ring type, incorrect results might arise due to overflow.

If new points are to be constructed, for example the intersection point of two lines, computation of Cartesian coordinates usually involves divisions. Hence, one needs to use a *FieldNumberType* with Cartesian representation, or alternatively, switch to homogeneous representation. The type *double* is a – though imprecise – model for *FieldNumberType*. You can also put any *RingNumberType* into the *Quotient* adaptor to get a field type which then can be put into *Cartesian*. But using homogeneous representation on the *RingNumberType* is usually the better option. Other valid *FieldNumberType*s are *leda_rational* and *leda_real*.

If it is crucial for you that the computation is reliable, the right choice is probably a number type that guarantees exact computation. The *Filtered_kernel* provides a way to apply filtering techniques [BBP01] to achieve a kernel with exact and efficient predicates. Still other people will prefer the built-in type `double`, because they need speed and can live with approximate results, or even algorithms that, from time to time, crash or compute incorrect results due to accumulated rounding errors.

Predefined kernels. For the user's convenience, CGAL provides 3 typedefs to generally useful kernels.

- They are all Cartesian kernels.
- They all support constructions of points from `double` Cartesian coordinates.
- All these 3 kernels provide exact geometric predicates.
- They handle geometric constructions differently:
  - *Exact_predicates_exact_constructions_kernel*: provides exact geometric constructions, in addition to exact geometric predicates.
  - *Exact_predicates_exact_constructions_kernel_with_sqrt*: same as *Exact_predicates_exact_constructions_kernel*, but the number type it provides (*Exact_predicates_exact_constructions_kernel_with_sqrt::FT*) supports the square root operation exactly [1].
  - *Exact_predicates_inexact_constructions_kernel*: provides exact geometric predicates, but geometric constructions may be inexact due to round-off errors. It is however enough for most CGAL algorithms, and faster than both *Exact_predicates_exact_constructions_kernel* and *Exact_predicates_exact_constructions_kernel_with_sqrt*.

[1] Currently it requires having either LEDA or CORE installed

11.3 Kernel Geometry

11.3.1 Points and Vectors

In CGAL, we strictly distinguish between points, vectors and directions. A *point* is a point in the Euclidean space $\mathbb{E}^d$, a *vector* is the difference of two points p_2, p_1 and denotes the direction and the distance from p_1 to p_2 in the vector space $\mathbb{R}^d$, and a *direction* is a vector where we forget about its length. They are different mathematical concepts. For example, they behave different under affine transformations and an addition of two points is meaningless in affine geometry. By putting them in different classes we not only get cleaner code, but also type checking by the compiler which avoids ambiguous expressions. Hence, it pays twice to make this distinction.

CGAL defines a symbolic constant *ORIGIN* of type *Origin* which denotes the point at the origin. This constant is used in the conversion between points and vectors. Subtracting it from a point p results in the locus vector of p.

```
Point_2< Cartesian<double> >  p(1.0, 1.0), q;
Vector_2< Cartesian<double> >  v;
v = p - ORIGIN;
q = ORIGIN + v;
assert( p == q );
```

In order to obtain the point corresponding to a vector v you simply have to add v to *ORIGIN*. If you want to determine the point q in the middle between two points p_1 and p_2, you can write[2]

```
q = p_1 + (p_2 - p_1) / 2.0;
```

Note that these constructions do not involve any performance overhead for the conversion with the currently available representation classes.

11.3.2 Kernel Objects

Besides points (*Point_2<Kernel>*, *Point_3<Kernel>*, *Point_d<Kernel>*), vectors (*Vector_2<Kernel>*, *Vector_3<Kernel>*), and directions (*Direction_2<Kernel>*, *Direction_3<Kernel>*), CGAL provides lines, rays, segments, planes, triangles, tetrahedra, iso-rectangles, iso-cuboids, circles and spheres.

Lines (*Line_2<Kernel>*, *Line_3<Kernel>*) in CGAL are oriented. In two-dimensional space, they induce a partition of the plane into a positive side and a negative side. Any two points on a line induce an orientation of this line. A ray (*Ray_2<Kernel>*, *Ray_3<Kernel>*) is semi-infinite interval on a line, and this line is oriented from the finite endpoint of this interval towards any other point in this interval. A segment (*Segment_2<Kernel>*, *Segment_3<Kernel>*) is a bounded interval on a directed line, and the endpoints are ordered so that they induce the same direction as that of the line.

Planes are affine subspaces of dimension two in $\mathbb{E}^3$, passing through three points, or a point and a line, ray, or segment. CGAL provides a correspondence between any plane in the ambient space $\mathbb{E}^3$ and the embedding of $\mathbb{E}^2$ in that space. Just like lines, planes are oriented and partition space into a positive side and a negative side. In CGAL, there are no special classes for half-spaces. Half-spaces in 2D and 3D are supposed to be represented by oriented lines and planes, respectively.

[2] you might call *midpoint(p_1,p_2)* instead

Concerning polygons and polyhedra, the kernel provides triangles, iso-oriented rectangles, iso-oriented cuboids and tetrahedra. More complex polygons[3] and polyhedra or polyhedral surfaces can be obtained from the basic library (*Polygon_2*, *Polyhedron_3*), so they are not part of the kernel. As with any Jordan curves, triangles, iso-oriented rectangles and circles separate the plane into two regions, one bounded and one unbounded.

11.3.3 Orientation and Relative Position

Geometric objects in CGAL have member functions that test the position of a point relative to the object. Full dimensional objects and their boundaries are represented by the same type, e.g. half-spaces and hyperplanes are not distinguished, neither are balls and spheres and discs and circles. Such objects split the ambient space into two full-dimensional parts, a bounded part and an unbounded part (e.g. circles), or two unbounded parts (e.g. hyperplanes). By default these objects are oriented, i.e., one of the resulting parts is called the positive side, the other one is called the negative side. Both of these may be unbounded.

For these objects there is a function *oriented_side()* that determines whether a test point is on the positive side, the negative side, or on the oriented boundary. These function returns a value of type *Oriented_side*.

Those objects that split the space in a bounded and an unbounded part, have a member function *bounded_side()* with return type *Bounded_side*.

If an object is lower dimensional, e.g. a triangle in three-dimensional space or a segment in two-dimensional space, there is only a test whether a point belongs to the object or not. This member function, which takes a point as an argument and returns a Boolean value, is called *has_on()*

11.4 Predicates and Constructions

11.4.1 Predicates

Predicates are at the heart of a geometry kernel. They are basic units for the composition of geometric algorithms and encapsulate decisions. Hence their correctness is crucial for the control flow and hence for the correctness of an implementation of a geometric algorithm. CGAL uses the term predicate in a generalized sense. Not only components returning a Boolean value are called predicates but also components returning an enumeration type like a *Comparison_result* or an *Orientation*. We say components, because predicates are implemented both as functions and function objects (provided by a kernel class).

CGAL provides predicates for the orientation of point sets (*orientation*, *leftturn*, *rightturn*, *collinear*, *coplanar*), for comparing points according to some given order, especially for comparing Cartesian coordinates (e.g. *lexicographically_xy_smaller*), in-circle and in-sphere tests, and predicates to compare distances.

11.4.2 Constructions

Functions and function objects that generate objects that are neither of type *bool* nor enum types are called constructions. Constructions involve computation of new numerical values and may be imprecise due to rounding errors unless a kernel with an exact number type is used.

[3] Any sequence of points can be seen as a (not necessary simple) polygon or polyline. This view is used frequently in the basic library as well.

Affine transformations (*Aff_transformation_2<Kernel>*, *Aff_transformation_3<Kernel>*) allow to generate new object instances under arbitrary affine transformations. These transformations include translations, rotations (in 2D only) and scaling. Most of the geometric objects in a kernel have a member function *transform(Aff_transformation t)* which applies the transformation to the object instance.

CGAL also provides a set of functions that detect or compute the intersection between objects of the 2D kernel, and many objects in the 3D kernel, and functions to calculate their squared distance. Moreover, some member functions of kernel objects are constructions.

So there are routines that compute the square of the Euclidean distance, but no routines that compute the distance itself. Why? First of all, the two values can be derived from each other quite easily (by taking the square root or taking the square). So, supplying only the one and not the other is only a minor inconvenience for the user. Second, often either value can be used. This is for example the case when (squared) distances are compared. Third, the library wants to stimulate the use of the squared distance instead of the distance. The squared distance can be computed in more cases and the computation is cheaper. We do this by not providing the perhaps more natural routine, The problem of a distance routine is that it needs the *sqrt* operation. This has two drawbacks:

- The *sqrt* operation can be costly. Even if it is not very costly for a specific number type and platform, avoiding it is always cheaper.
- There are number types on which no *sqrt* operation is defined, especially integer types and rationals.

11.4.3 Polymorphic Return Values

Some functions can return different types of objects. A typical C++ solution to this problem is to derive all possible return types from a common base class, to return a pointer to this class and to perform a dynamic cast on this pointer. The class *Object* provides an abstraction. An object *obj* of the class *Object* can represent an arbitrary class. The only operations it provides is to make copies and assignments, so that you can put them in lists or arrays. Note that *Object* is NOT a common base class for the elementary classes. Therefore, there is no automatic conversion from these classes to *Object*. Rather this is done with the global function *make_object()*. This encapsulation mechanism requires the use of *object_cast* to access the encapsulated class (a less efficient way, which is now discouraged, used to be to use the *assign* function).

Example

In the following example, the object class is used as return value for the intersection computation, as there are possibly different return values.

```
{
    typedef Cartesian<double>  K;
    typedef K::Point_2         Point_2;
    typedef K::Segment_2       Segment_2;

    Segment_2 segment_1, segment_2;

    std::cin >> segment_1 >> segment_2;

    Object obj = intersection(segment_1, segment_2);

    if (const Point_2 *point = object_cast<Point_2>(&obj)) {
        /* do something with *point */
    } else if (const Segment_2 *segment = object_cast<Segment_2>(&obj)) {
```

```
        /* do something with *segment*/
    }

    /* there was no intersection */
}
```

The intersection routine itself looks roughly as follows:

```
template < class Kernel >
Object  intersection(Segment_2<Kernel> s1, Segment_2<Kernel> s2)
{

    if (/* intersection in a point */ ) {

       Point_2<Kernel> p = ...  ;
       return make_object(p);

    } else if (/* intersection in a segment */ ) {

       Segment_2<Kernel> s = ...  ;
       return make_object(s);
    }
    return Object();
}
```

11.4.4 Constructive Predicates

For testing where a point p lies with respect to a plane defined by three points q, r and s, one may be tempted to construct the plane *Plane_3<Kernel>(q,r,s)* and use the method *oriented_side(p)*. This may pay off if many tests with respect to the plane are made. Nevertheless, unless the number type is exact, the constructed plane is only approximated, and round-off errors may lead *oriented_side(p)* to return an orientation which is different from the orientation of p, q, r, and s.

In CGAL, we provide predicates in which such geometric decisions are made directly with a reference to the input points p, q, r, s, without an intermediary object like a plane. For the above test, the recommended way to get the result is to use *orientation(p,q,r,s)*. For exact number types, the situation is different. If several tests are to be made with the same plane, it pays off to construct the plane and to use *oriented_side(p)*.

11.5 Extensible Kernel

This manual section describe how users can plug user defined geometric classes in existing CGAL kernels. This is best illustrated by an example.

11.5.1 Introduction

CGAL defines the concept of a geometry kernel. Such a kernel provides types, construction objects and generalized predicates. Most implementations of Computational Geometry algorithms and data structures in the basic library of CGAL were done in a way that classes or functions can be parametrized with a geometric traits class.

In most cases this geometric traits class must be a model of the CGAL geometry kernel concept (but there are some exceptions).

11.5.2 An Extensive Example

Assume you have the following point class, where the coordinates are stored in an array of *doubles*, where we have another data member *color*, which shows up in the constructor.

```
#ifndef MY_POINTC2_H
#define MY_POINTC2_H


#include <CGAL/Origin.h>
#include <CGAL/Bbox_2.h>


class MyPointC2 {

private:
  double vec[2];
  int col;

public:

  MyPointC2()
    : col(0)
  {
    *vec = 0;
    *(vec+1) = 0;
  }


  MyPointC2(const double x, const double y, int c = 0)
    : col(c)
  {
    *vec = x;
    *(vec+1) = y;
  }

  const double& x() const  { return *vec; }

  const double& y() const { return *(vec+1); }

  double & x() { return *vec; }

  double& y() { return *(vec+1); }
```

```
  int color() const { return col; }

  int& color() { return col; }


  bool operator==(const MyPointC2 &p) const
  {
    return ( *vec == *(p.vec) )  && ( *(vec+1) == *(p.vec + 1) && ( col
   == p.col) );
  }

  bool operator!=(const MyPointC2 &p) const
  {
      return !(*this == p);
  }

};

#endif // MY_POINTC2_H
```

File: examples/Kernel_23/MyPointC2.h

As said earlier the class is pretty minimalistic, for example it has no *bbox()* method. One might assume that a basic library algorithm which computes a bounding box (e.g, to compute the bounding box of a polygon), will not compile. Luckily it will, because it does not use of member functions of geometric objects, but it makes use of the functor *Kernel::Construct_bbox_2*.

To make the right thing happen with *MyPointC2* we have to provide the following functor.

```
#ifndef MYCONSTRUCT_BBOX_2_H
#define MYCONSTRUCT_BBOX_2_H


template <class ConstructBbox_2>
class MyConstruct_bbox_2 : public ConstructBbox_2 {
public:
  using ConstructBbox_2::operator();

  CGAL::Bbox_2 operator()(const MyPointC2& p) const {
    return CGAL::Bbox_2(p.x(), p.y(), p.x(), p.y());
  }
};

#endif //MYCONSTRUCT_BBOX_2_H
```

File: examples/Kernel_23/MyConstruct_bbox_2.h

Things are similar for random access to the Cartesian coordinates of a point. As the coordinates are stored in an array of *doubles* we can use *double** as random access iterator.

```
#ifndef MYCONSTRUCT_COORD_ITERATOR_H
```

```
#define MYCONSTRUCT_COORD_ITERATOR_H

class MyConstruct_coord_iterator {
public:
  const double* operator()(const MyPointC2& p)
  {
    return &p.x();
  }

  const double* operator()(const MyPointC2& p, int)
  {
    const double* pyptr = &p.y();
    pyptr++;
    return pyptr;
  }
};

#endif //MYCONSTRUCT_COORD_ITERATOR_H
```

File: examples/Kernel_23/MyConstruct_coord_iterator.h

The last functor we have to provide is the one which constructs points. That is you are not forced to add the constructor with the *Origin* as parameter to your class, nor the constructor with homogeneous coordinates. The functor is a kind of glue layer between the CGAL algorithms and your class.

```
#ifndef MYCONSTRUCT_POINT_2_H
#define MYCONSTRUCT_POINT_2_H

template <typename K, typename OldK>
class MyConstruct_point_2
{
  typedef typename K::RT         RT;
  typedef typename K::Point_2    Point_2;
  typedef typename K::Line_2     Line_2;
  typedef typename Point_2::Rep  Rep;
public:
  typedef Point_2                result_type;

  // Note : the CGAL::Return_base_tag is really internal CGAL stuff.
  // Unfortunately it is needed for optimizing away copy-constructions,
  // due to current lack of delegating constructors in the C++ standard.
  Rep // Point_2
  operator()(CGAL::Return_base_tag, CGAL::Origin o) const
  { return Rep(o); }

  Rep // Point_2
  operator()(CGAL::Return_base_tag, const RT& x, const RT& y) const
  { return Rep(x, y); }

  Rep // Point_2
  operator()(CGAL::Return_base_tag, const RT& x, const RT& y, const RT&
   w) const
  { return Rep(x, y, w); }
```

```
  Point_2
  operator()(CGAL::Origin o) const
  { return MyPointC2(0, 0, 0); }

  Point_2
  operator()(const RT& x, const RT& y) const
  {
    return MyPointC2(x, y, 0);
  }

  Point_2
  operator()(const Line_2& l) const
  {
    typename OldK::Construct_point_2 base_operator;
    Point_2 p = base_operator(l);
    return p;
  }

  Point_2
  operator()(const Line_2& l, int i) const
  {
    typename OldK::Construct_point_2 base_operator;
    return base_operator(l, i);
  }

  // We need this one, as such a functor is in the Filtered_kernel
  Point_2
  operator()(const RT& x, const RT& y, const RT& w) const
  {
    if(w != 1){
      return MyPointC2(x/w, y/w, 0);
    } else {
      return MyPointC2(x,y, 0);
    }
  }
};

#endif //MYCONSTRUCT_POINT_2_H
```

File: examples/Kernel_23/MyConstruct_point_2.h

Now we are ready to put the puzzle together. We won't explain it in detail, but you see that there are *typedefs* to the new point class and the functors. All the other types are inherited.

```
#ifndef MYKERNEL_H
#define MYKERNEL_H

#include <CGAL/Cartesian.h>
#include "MyPointC2.h"
#include "MySegmentC2.h"
#include "MyConstruct_bbox_2.h"
#include "MyConstruct_coord_iterator.h"
```

```
#include "MyConstruct_point_2.h"

// K_ is the new kernel, and K_Base is the old kernel
template < typename K_, typename K_Base >
class MyCartesian_base
  : public K_Base::template Base<K_>::Type
{
  typedef typename K_Base::template Base<K_>::Type   OldK;
public:
  typedef K_                                  Kernel;
  typedef MyPointC2                           Point_2;
  typedef MySegmentC2<Kernel>                 Segment_2;
  typedef MyConstruct_point_2<Kernel, OldK>         Construct_point_2;
  typedef const double*                       Cartesian_const_iterator_2;
  typedef MyConstruct_coord_iterator
   Construct_cartesian_const_iterator_2;
  typedef MyConstruct_bbox_2<typename OldK::Construct_bbox_2>
                                              Construct_bbox_2;

  Construct_point_2
  construct_point_2_object() const
  { return Construct_point_2(); }

  Construct_bbox_2
  construct_bbox_2_object() const
  { return Construct_bbox_2(); }

  Construct_cartesian_const_iterator_2
  construct_cartesian_const_iterator_2_object() const
  { return Construct_cartesian_const_iterator_2(); }

  template < typename Kernel2 >
  struct Base { typedef MyCartesian_base<Kernel2, K_Base>  Type; };
};


template < typename FT_ >
struct MyKernel
  : public CGAL::Type_equality_wrapper<
                MyCartesian_base<MyKernel<FT_>, CGAL::Cartesian<FT_> >,
                MyKernel<FT_> >
{};

#endif // MYKERNEL_H
```

File: examples/Kernel_23/MyKernel.h

Finally, we give an example how this new kernel can be used. Predicates and constructions work with the new point, they can be a used to construct segments and triangles with, and data structures from the Basic Library, as the Delaunay triangulation work with them.

The kernel itself can be made robust by plugging it in the *Filtered_kernel*.

```
#include <CGAL/basic.h>
#include <CGAL/Filtered_kernel.h>
#include <CGAL/Delaunay_triangulation_2.h>
#include <CGAL/squared_distance_2.h>
#include <cassert>
#include "MyKernel.h"
#include "MyPointC2_iostream.h"

typedef MyKernel<double>                   MK;
typedef CGAL::Filtered_kernel_adaptor<MK>  K;
typedef CGAL::Delaunay_triangulation_2<K>  Delaunay_triangulation_2;


typedef K::Point_2         Point;
typedef K::Segment_2       Segment;
typedef K::Ray_2           Ray;
typedef K::Line_2          Line;
typedef K::Triangle_2      Triangle;
typedef K::Iso_rectangle_2 Iso_rectangle;

const int RED= 1;
const int BLACK=2;

int main()
{
  Point a(0,0), b(1,0), c(1,1), d(0,1);
  a.color()=RED;
  b.color()=BLACK;
  d.color()=RED;

  Delaunay_triangulation_2 dt;
  dt.insert(a);

  K::Orientation_2 orientation;
  orientation(a,b,c);

  Point p(1,2), q;
  p.color() = RED;
  q.color() = BLACK;
  std::cout << p << std::endl;

  K::Compute_squared_distance_2 squared_distance;

  std::cout << "squared_distance(a, b) == "
            << squared_distance(a, b) << std::endl;

  Segment s1(p,q), s2(a, c);

  K::Construct_midpoint_2 construct_midpoint_2;

  Point mp = construct_midpoint_2(p,q);

  assert(s1.source().color() == RED);
```

```
  K::Intersect_2 intersection;

  CGAL::Object o = intersection(s1, s2);

  K::Construct_cartesian_const_iterator_2 construct_it;
  K::Cartesian_const_iterator_2  cit = construct_it(a);
  assert(*cit == a.x());

  cit = construct_it(a,0);

  cit--;
  assert(*cit == a.y());

  Line l1(a,b), l2(p, q);

  intersection(l1, l2);


  intersection(s1, l1);

  Ray r1(d,b), r2(d,c);
  intersection(r1, r2);

  intersection(r1, l1);

  squared_distance(r1, r2);
  squared_distance(r1, l2);
  squared_distance(r1, s2);

  Triangle t1(a,b,c), t2(a,c,d);
  intersection(t1, t2);
  intersection(t1, l1);

  intersection(t1, s1);

  intersection(t1, r1);

  Iso_rectangle i1(a,c), i2(d,p);
  intersection(i1, i2);
  intersection(i1, s1);

  intersection(i1, r1);
  intersection(i1, l1);

  t1.orientation();

  std::cout << s1.source() << std::endl;

  std::cout << t1.bbox() << std::endl;

  std::cout << "done" << std::endl;
  return 0;
}
```

File: `examples/Kernel_23/MyKernel.cpp`

11.5.3 Limitations

The point class must have member functions *x()* and *y()* (and *z()* for the 3d point). We will probably introduce function objects that take care of coordinate access.

As we enforce type equality between *MyKernel::Point_2* and *Point_2<MyKernel>*, the constructor with the color as third argument is not available.

11.6 Projection Traits Classes

It is sometimes useful to apply 2D algorithms to the projection of 3D points on a plane. Examples are triangulated terrains, which are points with elevation, or surface reconstruction from parallel slices, where one wants to check the simplicity or orientation of polygons.

For this purpose CGAL provides several projection traits classes, which are a model of traits class concepts of 2D triangulations, 2D polygon and 2D convex hull traits classes. The projection traits classes are listed in the "*Is Model for the Concepts*" sections of the concepts.

11.7 Design and Implementation History

At a meeting at Utrecht University in January 1995, Olivier Devillers, Andreas Fabri, Wolfgang Freiseisen, Geert-Jan Giezeman, Mark Overmars, Stefan Schirra, Otfried Schwarzkopf (now Otfried Cheong), and Sven Schönherr discussed the foundations of the CGAL kernel. Many design and software engineering issues were addressed, e.g. naming conventions, coupling of classes (flat versus deep class hierarchy), memory allocation, programming conventions, mutability of atomic objects, points and vectors, storing additional information, orthogonality of operations on the kernel objects, viewing non-constant-size objects like polygons as dynamic data structures (and hence not as part of the (innermost) kernel).

The people attending the meeting delegated the compilation of a draft specification to Stefan Schirra. The resulting draft specification was intentionally modeled on CGAL's precursors C++GAL and PLAGEO as well as on the geometric part of LEDA. The specification already featured coexistence of Cartesian and homogeneous representation of point/vector data and parameterization by number type(s). During the discussion of the draft a kernel design group was formed. The members of this group were Andreas Fabri, Geert-Jan Giezeman, Lutz Kettner, Stefan Schirra, and Sven Schönherr. The work of the kernel design group led to significant changes and improvements of the original design, e.g. the strong separation between points and vectors. Probably the most important enhancement was the design of a common superstructure for the previously uncoupled Cartesian and homogeneous representations. One can say, that the kernel was designed by this group. The kernel was later revised based on suggestions by Hervé Brönnimann, Bernd Gärtner, Michael Hoffmann, and Lutz Kettner.

A first version of the kernel was internally made available at the beginning of the CGAL-project (ESPRIT LTR IV project number 21957). Since then many more people contributed to the evolution of the kernel through discussions on the CGAL mailing lists. The implementation based on Cartesian representation was (initially) provided by Andreas Fabri, the homogeneous representation (initially) by Stefan Schirra. Intersection and distance computations were implemented by Geert-Jan Giezeman. Further work has been done by Susan Hert on the overall maintenance of the kernel. Philippe Guigue has provided efficient intersection tests for 3D triangles. Andreas Fabri, Michael Hoffmann and Sylvain Pion have improved the support for the extensibility and adaptability of the kernel. Pedro Machado Manhães de Castro and Monique Teillaud introduced 3D circles. In 2010, Pierre Alliez, Stéphane Tayeb and Camille Wormser added intersection constructions for 3D triangles and efficient intersection tests for bounding boxes.

11.7.1 Acknowledgment

This work was supported by the Graduiertenkolleg 'Algorithmische Diskrete Mathematik', under grant DFG We 1265/2-1, and by ESPRIT IV Long Term Research Projects No. 21957 (CGAL) and No. 28155 (GALIA).

2D and 3D Geometry Kernel Reference Manual

Hervé Brönnimann, Andreas Fabri, Geert-Jan Giezeman, Susan Hert, Michael Hoffmann, Lutz Kettner, Sylvain Pion, and Stefan Schirra

The following pages give a complete overview on the functionality provided in the kernel.

Concepts

Kernel Classes and Operations

Predefined Kernels

Classes for 2D Geometry

Classes for 3D Geometry

Classes for dD Geometry

Constants and Enumerations

Distance and Intersection Operations

Predicates and Constructions

11.8 Concepts

Concept

Kernel

The concept of a *kernel* is defined by a set of requirements on the provision of certain types and access member functions to create objects of these types. The types are function object classes to be used within the algorithms and data structures of CGAL. This allows you to use any model of a kernel as a traits class in the CGAL algorithms and data structures, unless they require types beyond those provided by a kernel.

A kernel provides types, construction objects, and generalized predicates. The former replace constructors of the kernel classes and constructive procedures in the kernel. There are also function objects replacing operators, especially for equality testing.

Has Models

CGAL::Cartesian<FieldNumberType> . . . . page 588
CGAL::Homogeneous<RingNumberType> . . . . page 1070
CGAL::Simple_cartesian<FieldNumberType> . . . . page 602
CGAL::Simple_homogeneous<RingNumberType> . . . . page 603
CGAL::Filtered_kernel<CK> . . . . page 592
CGAL::Exact_predicates_exact_constructions_kernel . . . . page 606
CGAL::Exact_predicates_exact_constructions_kernel_with_sqrt . . . . page 607
CGAL::Exact_predicates_inexact_constructions_kernel . . . . page 608

Types

Kernel:: FT	a model of *FieldNumberType*
Kernel:: RT	a model of *RingNumberType*

The following types describe the return types of predicates. They typically map to *bool* and CGAL kernel enum types, except when an interval arithmetic number type is used such as within the filtering kernels, in which case it is *Uncertain<bool>* or similar.

Kernel:: Boolean	*bool* or *Uncertain<bool>*
Kernel:: Sign	*CGAL::Sign* or *Uncertain<CGAL::Sign>*
Kernel:: Comparison_result	*CGAL::Comparison_result* or *Uncertain< CGAL::Comparison_result>*
Kernel:: Orientation	*CGAL::Orientation* or *Uncertain<CGAL::Orientation>*
Kernel:: Oriented_side	*CGAL::Oriented_side* or *Uncertain<CGAL::Oriented_side>*
Kernel:: Bounded_side	*CGAL::Bounded_side* or *Uncertain<CGAL::Bounded_side>*
Kernel:: Angle	*CGAL::Angle* or *Uncertain<CGAL::Angle>*

Constants

static const bool	*Has_filtered_predicates;*

A Boolean value indicating whether the predicates are filtered (as in *Filtered_kernel*). This helps propagating such decisions to traits classes which are built on top of a kernel, so that they can decide to filter their own predicates or not.

Coordinate Access

Kernel:: Cartesian_const_iterator_2	a model of *Kernel::CartesianConstIterator_2*

Geometric Objects

Kernel:: Point_2	a model of *Kernel::Point_2*
Kernel:: Vector_2	a model of *Kernel::Vector_2*
Kernel:: Direction_2	a model of *Kernel::Direction_2*
Kernel:: Line_2	a model of *Kernel::Line_2*
Kernel:: Ray_2	a model of *Kernel::Ray_2*
Kernel:: Segment_2	a model of *Kernel::Segment_2*
Kernel:: Triangle_2	a model of *Kernel::Triangle_2*
Kernel:: Iso_rectangle_2	a model of *Kernel::IsoRectangle_2*
Kernel:: Circle_2	a model of *Kernel::Circle_2*
Kernel:: Object_2	a model of *Kernel::Object_2*

Constructions

Kernel:: Construct_point_2	a model of *Kernel::ConstructPoint_2*
Kernel:: Construct_vector_2	a model of *Kernel::ConstructVector_2*
Kernel:: Construct_direction_2	a model of *Kernel::ConstructDirection_2*
Kernel:: Construct_segment_2	a model of *Kernel::ConstructSegment_2*
Kernel:: Construct_line_2	a model of *Kernel::ConstructLine_2*
Kernel:: Construct_ray_2	a model of *Kernel::ConstructRay_2*
Kernel:: Construct_circle_2	a model of *Kernel::ConstructCircle_2*
Kernel:: Construct_triangle_2	a model of *Kernel::ConstructTriangle_2*
Kernel:: Construct_iso_rectangle_2	a model of *Kernel::ConstructIsoRectangle_2*
Kernel:: Construct_object_2	a model of *Kernel::ConstructObject_2*
Kernel:: Construct_scaled_vector_2	a model of *Kernel::ConstructScaledVector_2*
Kernel:: Construct_translated_point_2	a model of *Kernel::ConstructTranslatedPoint_2*
Kernel:: Construct_point_on_2	a model of *Kernel::ConstructPointOn_2*
Kernel:: Construct_projected_point_2	a model of *Kernel::ConstructProjectedPoint_2*
Kernel:: Construct_projected_xy_point_2	a model of *Kernel::ConstructProjectedXYPoint_2*
Kernel:: Construct_cartesian_const_iterator_2	a model of *Kernel::ConstructCartesianConstIterator_2*
Kernel:: Construct_vertex_2	a model of *Kernel::ConstructVertex_2*
Kernel:: Construct_bbox_2	a model of *Kernel::ConstructBbox_2*
Kernel:: Construct_perpendicular_vector_2	a model of *Kernel::ConstructPerpendicularVector_2*
Kernel:: Construct_perpendicular_direction_2	a model of *Kernel::ConstructPerpendicularDirection_2*
Kernel:: Construct_perpendicular_line_2	a model of *Kernel::ConstructPerpendicularLine_2*
Kernel:: Construct_max_vertex_2	a model of *Kernel::ConstructMaxVertex_2*
Kernel:: Construct_midpoint_2	a model of *Kernel::ConstructMidpoint_2*

Kernel:: Construct_equidistant_line_3	a model of *Kernel::ConstructEquidistantLine_3*
Kernel:: Construct_min_vertex_2	a model of *Kernel::ConstructMinVertex_2*
Kernel:: Construct_center_2	a model of *Kernel::ConstructCenter_2*
Kernel:: Construct_centroid_2	a model of *Kernel::ConstructCentroid_2*
Kernel:: Construct_circumcenter_2	a model of *Kernel::ConstructCircumcenter_2*
Kernel:: Construct_bisector_2	a model of *Kernel::ConstructBisector_2*
Kernel:: Construct_opposite_direction_2	a model of *Kernel::ConstructOppositeDirection_2*
Kernel:: Construct_opposite_segment_2	a model of *Kernel::ConstructOppositeSegment_2*
Kernel:: Construct_opposite_ray_2	a model of *Kernel::ConstructOppositeRay_2*
Kernel:: Construct_opposite_line_2	a model of *Kernel::ConstructOppositeLine_2*
Kernel:: Construct_opposite_triangle_2	a model of *Kernel::ConstructOppositeTriangle_2*
Kernel:: Construct_opposite_circle_2	a model of *Kernel::ConstructOppositeCircle_2*
Kernel:: Construct_opposite_vector_2	a model of *Kernel::ConstructOppositeVector_2*

If the result type is not determined, there is no *Construct_* prefix:

Kernel:: Intersect_2	a model of *Kernel::Intersect_2*
Kernel:: Assign_2	a model of *Kernel::Assign_2*

If the result type is a number type, the prefix is *Compute_*:

Kernel:: Compute_squared_distance_2	a model of *Kernel::ComputeSquaredDistance_2*
Kernel:: Compute_squared_length_2	a model of *Kernel::ComputeSquaredLength_2*
Kernel:: Compute_squared_radius_2	a model of *Kernel::ComputeSquaredRadius_2*
Kernel:: Compute_area_2	a model of *Kernel::ComputeArea_2*
Kernel:: Compute_determinant_2	a model of *Kernel::ComputeDeterminant_2*

Generalized Predicates

Kernel:: Angle_2	a model of *Kernel::Angle_2*
Kernel:: Equal_2	a model of *Kernel::Equal_2*
Kernel:: Equal_x_2	a model of *Kernel::EqualX_2*
Kernel:: Equal_y_2	a model of *Kernel::EqualY_2*
Kernel:: Less_x_2	a model of *Kernel::LessX_2*
Kernel:: Less_y_2	a model of *Kernel::LessY_2*
Kernel:: Less_xy_2	a model of *Kernel::LessXY_2*
Kernel:: Less_yx_2	a model of *Kernel::LessYX_2*
Kernel:: Compare_x_2	a model of *Kernel::CompareX_2*
Kernel:: Compare_x_at_y_2	a model of *Kernel::CompareXAtY_2*
Kernel:: Compare_y_2	a model of *Kernel::CompareY_2*
Kernel:: Compare_xy_2	a model of *Kernel::CompareXY_2*
Kernel:: Compare_yx_2	a model of *Kernel::CompareYX_2*
Kernel:: Compare_y_at_x_2	a model of *Kernel::CompareYAtX_2*
Kernel:: Compare_distance_2	a model of *Kernel::CompareDistance_2*
Kernel:: Compare_angle_with_x_axis_2	a model of *Kernel::CompareAngleWithXAxis_2*
Kernel:: Compare_slope_2	a model of *Kernel::CompareSlope_2*
Kernel:: Less_distance_to_point_2	a model of *Kernel::LessDistanceToPoint_2*
Kernel:: Less_signed_distance_to_line_2	a model of *Kernel::LessSignedDistanceToLine_2*
Kernel:: Less_rotate_ccw_2	a model of *Kernel::LessRotateCCW_2*
Kernel:: Left_turn_2	a model of *Kernel::LeftTurn_2*
Kernel:: Collinear_2	a model of *Kernel::Collinear_2*
Kernel:: Orientation_2	a model of *Kernel::Orientation_2*
Kernel:: Side_of_oriented_circle_2	a model of *Kernel::SideOfOrientedCircle_2*

Kernel:: Side_of_bounded_circle_2	a model of *Kernel::SideOfBoundedCircle_2*
Kernel:: Is_horizontal_2	a model of *Kernel::IsHorizontal_2*
Kernel:: Is_vertical_2	a model of *Kernel::IsVertical_2*
Kernel:: Is_degenerate_2	a model of *Kernel::IsDegenerate_2*
Kernel:: Has_on_2	a model of *Kernel::HasOn_2*
Kernel:: Collinear_has_on_2	a model of *Kernel::CollinearHasOn_2*
Kernel:: Has_on_bounded_side_2	a model of *Kernel::HasOnBoundedSide_2*
Kernel:: Has_on_unbounded_side_2	a model of *Kernel::HasOnUnboundedSide_2*
Kernel:: Has_on_boundary_2	a model of *Kernel::HasOnBoundary_2*
Kernel:: Has_on_positive_side_2	a model of *Kernel::HasOnPositiveSide_2*
Kernel:: Has_on_negative_side_2	a model of *Kernel::HasOnNegativeSide_2*
Kernel:: Oriented_side_2	a model of *Kernel::OrientedSide_2*
Kernel:: Bounded_side_2	a model of *Kernel::BoundedSide_2*
Kernel:: Are_parallel_2	a model of *Kernel::AreParallel_2*
Kernel:: Are_ordered_along_line_2	a model of *Kernel::AreOrderedAlongLine_2*
Kernel:: Are_strictly_ordered_along_line_2	a model of *Kernel::AreStrictlyOrderedAlongLine_2*
Kernel:: Collinear_are_ordered_along_line_2	a model of *Kernel::CollinearAreOrderedAlongLine_2*
Kernel:: Collinear_are_strictly_ordered_along_line_2	a model of *Kernel::CollinearAreStrictlyOrderedAlongLine_2*
Kernel:: Counterclockwise_in_between_2	a model of *Kernel::CounterclockwiseInBetween_2*
Kernel:: Do_intersect_2	a model of *Kernel::DoIntersect_2*

Three-dimensional Kernel

Coordinate Access

Kernel:: Cartesian_const_iterator_3	a model of *Kernel::CartesianConstIterator_3*

Geometric Objects

Kernel:: Point_3	a model of *Kernel::Point_3*
Kernel:: Vector_3	a model of *Kernel::Vector_3*
Kernel:: Direction_3	a model of *Kernel::Direction_3*
Kernel:: Iso_cuboid_3	a model of *Kernel::IsoCuboid_3*
Kernel:: Line_3	a model of *Kernel::Line_3*
Kernel:: Ray_3	a model of *Kernel::Ray_3*
Kernel:: Circle_3	a model of *Kernel::Circle_3*
Kernel:: Sphere_3	a model of *Kernel::Sphere_3*
Kernel:: Segment_3	a model of *Kernel::Segment_3*
Kernel:: Plane_3	a model of *Kernel::Plane_3*
Kernel:: Triangle_3	a model of *Kernel::Triangle_3*
Kernel:: Tetrahedron_3	a model of *Kernel::Tetrahedron_3*
Kernel:: Object_3	a model of *Kernel::Object_3*

Constructions

Kernel:: Construct_point_3	a model of *Kernel::ConstructPoint_3*
Kernel:: Construct_vector_3	a model of *Kernel::ConstructVector_3*

Kernel:: Construct_direction_3	a model of *Kernel::ConstructDirection_3*
Kernel:: Construct_plane_3	a model of *Kernel::ConstructPlane_3*
Kernel:: Construct_iso_cuboid_3	a model of *Kernel::ConstructIsoCuboid_3*
Kernel:: Construct_line_3	a model of *Kernel::ConstructLine_3*
Kernel:: Construct_ray_3	a model of *Kernel::ConstructRay_3*
Kernel:: Construct_sphere_3	a model of *Kernel::ConstructSphere_3*
Kernel:: Construct_segment_3	a model of *Kernel::ConstructSegment_3*
Kernel:: Construct_triangle_3	a model of *Kernel::ConstructTriangle_3*
Kernel:: Construct_tetrahedron_3	a model of *Kernel::ConstructTetrahedron_3*
Kernel:: Construct_object_3	a model of *Kernel::ConstructObject_3*
Kernel:: Construct_scaled_vector_3	a model of *Kernel::ConstructScaledVector_3*
Kernel:: Construct_translated_point_3	a model of *Kernel::ConstructTranslatedPoint_3*
Kernel:: Construct_point_on_3	a model of *Kernel::ConstructPointOn_3*
Kernel:: Construct_projected_point_3	a model of *Kernel::ConstructProjectedPoint_3*
Kernel:: Construct_lifted_point_3	a model of *Kernel::ConstructLiftedPoint_3*
Kernel:: Construct_cartesian_const_iterator_3	a model of *Kernel::ConstructCartesianConstIterator_3*
Kernel:: Construct_vertex_3	a model of *Kernel::ConstructVertex_3*
Kernel:: Construct_bbox_3	a model of *Kernel::ConstructBbox_3*
Kernel:: Construct_supporting_plane_3	a model of *Kernel::ConstructSupportingPlane_3*
Kernel:: Construct_orthogonal_vector_3	a model of *Kernel::ConstructOrthogonalVector_3*
Kernel:: Construct_base_vector_3	a model of *Kernel::ConstructBaseVector_3*
Kernel:: Construct_perpendicular_plane_3	a model of *Kernel::ConstructPerpendicularPlane_3*
Kernel:: Construct_radical_plane_3	a model of *Kernel::ConstructRadicalPlane_3*
Kernel:: Construct_perpendicular_line_3	a model of *Kernel::ConstructPerpendicularLine_3*
Kernel:: Construct_midpoint_3	a model of *Kernel::ConstructMidpoint_3*
Kernel:: Construct_center_3	a model of *Kernel::ConstructCenter_3*
Kernel:: Construct_centroid_3	a model of *Kernel::ConstructCentroid_3*
Kernel:: Construct_circumcenter_3	a model of *Kernel::ConstructCircumcenter_3*
Kernel:: Construct_bisector_3	a model of *Kernel::ConstructBisector_3*
Kernel:: Construct_cross_product_vector_3	a model of *Kernel::ConstructCrossProductVector_3*
Kernel:: Construct_opposite_direction_3	a model of *Kernel::ConstructOppositeDirection_3*
Kernel:: Construct_opposite_segment_3	a model of *Kernel::ConstructOppositeSegment_3*
Kernel:: Construct_opposite_ray_3	a model of *Kernel::ConstructOppositeRay_3*
Kernel:: Construct_opposite_line_3	a model of *Kernel::ConstructOppositeLine_3*
Kernel:: Construct_opposite_plane_3	a model of *Kernel::ConstructOppositePlane_3*
Kernel:: Construct_opposite_sphere_3	a model of *Kernel::ConstructOppositeSphere_3*
Kernel:: Construct_opposite_vector_3	a model of *Kernel::ConstructOppositeVector_3*

If the result type is not determined, there is no *Construct_* prefix:

Kernel:: Intersect_3	a model of *Kernel::Intersect_3*
Kernel:: Assign_3	a model of *Kernel::Assign_3*

If the result type is a number type, the prefix is *Compute_*:

Kernel:: Compute_area_3	a model of *Kernel::ComputeArea_3*
Kernel:: Compute_squared_area_3	a model of *Kernel::ComputeSquaredArea_3*
Kernel:: Compute_area_divided_by_pi_3	a model of *Kernel::ComputeAreaDividedByPi_3*
Kernel:: Compute_approximate_area_3	a model of *Kernel::ComputeApproximateArea_3*
Kernel:: Compute_determinant_3	a model of *Kernel::ComputeDeterminant_3*
Kernel:: Compute_squared_distance_3	a model of *Kernel::ComputeSquaredDistance_3*
Kernel:: Compute_squared_length_3	a model of *Kernel::ComputeSquaredLength_3*

Kernel:: Compute_squared_length_divided_by_pi_square_3	a model of *Kernel::ComputeSquaredLengthDividedByPiSquare_3*
Kernel:: Compute_approximate_squared_length_3	a model of *Kernel::ComputeApproximateSquaredLength_3*
Kernel:: Compute_squared_radius_3	a model of *Kernel::ComputeSquaredRadius_3*
Kernel:: Compute_volume_3	a model of *Kernel::ComputeVolume_3*

Generalized Predicates

Kernel:: Angle_3	a model of *Kernel::Angle_3*
Kernel:: Equal_3	a model of *Kernel::Equal_3*
Kernel:: Equal_x_3	a model of *Kernel::EqualX_3*
Kernel:: Equal_y_3	a model of *Kernel::EqualY_3*
Kernel:: Equal_z_3	a model of *Kernel::EqualZ_3*
Kernel:: Equal_xy_3	a model of *Kernel::EqualXY_3*
Kernel:: Less_x_3	a model of *Kernel::LessX_3*
Kernel:: Less_y_3	a model of *Kernel::LessY_3*
Kernel:: Less_z_3	a model of *Kernel::LessZ_3*
Kernel:: Less_xy_3	a model of *Kernel::LessXY_3*
Kernel:: Less_xyz_3	a model of *Kernel::LessXYZ_3*
Kernel:: Compare_x_3	a model of *Kernel::CompareX_3*
Kernel:: Compare_y_3	a model of *Kernel::CompareY_3*
Kernel:: Compare_z_3	a model of *Kernel::CompareZ_3*
Kernel:: Compare_xy_3	a model of *Kernel::CompareXY_3*
Kernel:: Compare_xyz_3	a model of *Kernel::CompareXYZ_3*
Kernel:: Less_signed_distance_to_plane_3	a model of *Kernel::LessSignedDistanceToPlane_3*
Kernel:: Less_distance_to_point_3	a model of *Kernel::LessDistanceToPoint_3*
Kernel:: Compare_dihedral_angle_3	a model of *Kernel::CompareDihedralAngle_3*
Kernel:: Compare_distance_3	a model of *Kernel::CompareDistance_3*
Kernel:: Collinear_3	a model of *Kernel::Collinear_3*
Kernel:: Coplanar_3	a model of *Kernel::Coplanar_3*
Kernel:: Orientation_3	a model of *Kernel::Orientation_3*
Kernel:: Coplanar_orientation_3	a model of *Kernel::CoplanarOrientation_3*
Kernel:: Coplanar_side_of_bounded_circle_3	a model of *Kernel::CoplanarSideOfBoundedCircle_3*
Kernel:: Side_of_oriented_sphere_3	a model of *Kernel::SideOfOrientedSphere_3*
Kernel:: Side_of_bounded_sphere_3	a model of *Kernel::SideOfBoundedSphere_3*
Kernel:: Is_degenerate_3	a model of *Kernel::IsDegenerate_3*
Kernel:: Has_on_3	a model of *Kernel::HasOn_3*
Kernel:: Has_on_bounded_side_3	a model of *Kernel::HasOnBoundedSide_3*
Kernel:: Has_on_unbounded_side_3	a model of *Kernel::HasOnUnboundedSide_3*
Kernel:: Has_on_boundary_3	a model of *Kernel::HasOnBoundary_3*
Kernel:: Has_on_positive_side_3	a model of *Kernel::HasOnPositiveSide_3*
Kernel:: Has_on_negative_side_3	a model of *Kernel::HasOnNegativeSide_3*
Kernel:: Oriented_side_3	a model of *Kernel::OrientedSide_3*
Kernel:: Bounded_side_3	a model of *Kernel::BoundedSide_3*
Kernel:: Are_parallel_3	a model of *Kernel::AreParallel_3*
Kernel:: Are_ordered_along_line_3	a model of *Kernel::AreOrderedAlongLine_3*
Kernel:: Are_strictly_ordered_along_line_3	a model of *Kernel::AreStrictlyOrderedAlongLine_3*
Kernel:: Collinear_are_ordered_along_line_3	a model of *Kernel::CollinearAreOrderedAlongLine_3*

Kernel:: Collinear_are_strictly_ordered_along_line_3	
	a model of *Kernel::CollinearAreStrictlyOrderedAlongLine_3*
Kernel:: Do_intersect_3	a model of *Kernel::DoIntersect_3*

Operations

For each of the function objects above, there must exist a member function that requires no arguments and returns an instance of that function object. The name of the member function is the uncapitalized name of the type returned with the suffix *_object* appended. For example, for the function object *Kernel::Construct_vector_2* the following member function must exist:

Kernel::Construct_vector_2	*kernel.construct_vector_2_object() const*

See Also
Kernel_d

Class

11.9 Kernel Classes and Operations

CGAL::Cartesian<FieldNumberType>

#include <CGAL/Cartesian.h>

Definition

A model for *Kernel* that uses Cartesian coordinates to represent the geometric objects. In order for *Cartesian<FieldNumberType>* to model Euclidean geometry in E^2 and/or E^3, for some mathematical field E (*e.g.*, the rationals $\mathbb{Q}$ or the reals $\mathbb{R}$), the template parameter *FieldNumberType* must model the mathematical field E. That is, the field operations on this number type must compute the mathematically correct results. If the number type provided as a model for *FieldNumberType* is only an approximation of a field (such as the built-in type *double*), then the geometry provided by the kernel is only an approximation of Euclidean geometry.

Is Model for the Concepts

Kernel . . . . page 581

Types

typedef FieldNumberType *FT;*
typedef FieldNumberType *RT;*

Implementation

All geometric objects in *Cartesian<FieldNumberType>* are reference counted.

See Also

CGAL::Simple_cartesian<FieldNumberType> . . . . page 602
CGAL::Homogeneous<RingNumberType> . . . . page 1070
CGAL::Simple_homogeneous<RingNumberType> . . . . page 603

CGAL::Cartesian_converter<K1, K2, NTConverter>

Class

Definition

Cartesian_converter<K1, K2, NTConverter> converts objects from the kernel traits *K1* to the kernel traits *K2* using *Converter* to do the conversion. Those traits must be of the form *Cartesian<FT1>* and *Cartesian<FT2>* (or the equivalent with *Simple_cartesian*). It then provides the following operators to convert objects from *K1* to *K2*.

The third template parameter *NTConverter* is a function object that must provide *K2::FT operator()(K1::FT n)* that converts *n* to an *K2::FT* which has the same value.

The default value of this parameter is *CGAL::NT_converter<K1::FT, K2::FT>*.

#include <CGAL/Cartesian_converter.h>

Creation

Cartesian_converter<K1, K2, NTConverter> conv;

Default constructor.

Operations

K2::Point_2 *conv.operator()(K1::Point_2 p)*

returns a *K2::Point_2* which coordinates are those of *p*, converted by *NTConverter*.

Similar operators are defined for the other kernel traits types *Point_3*, *Vector_2*...

Example

In the following example, we compute exactly the intersection point between a line and a triangle, and we then create a double approximation of this point.

```
#include <CGAL/Simple_cartesian.h>
#include <CGAL/Quotient.h>
#include <CGAL/MP_Float.h>
#include <CGAL/Cartesian_converter.h>

typedef CGAL::Simple_cartesian<double>                        IK;
typedef CGAL::Simple_cartesian<CGAL::Quotient<CGAL::MP_Float> >  EK;
typedef CGAL::Cartesian_converter<IK,EK>
   IK_to_EK;
typedef CGAL::Cartesian_converter<EK,IK>
   EK_to_IK;
```

```
int main(){
  IK::Triangle_3 t1(
    IK::Point_3(0.,0.,0.),
    IK::Point_3(1.,0.,-1.),
    IK::Point_3(0.,1.,3.)
  );

  IK::Line_3 l1(
    IK::Point_3(0.2,0.25,-7),
    IK::Point_3(0.25,0.3,4)
  );

  IK_to_EK to_exact;

  EK::Triangle_3 t2=to_exact(t1);
  EK::Line_3     l2=to_exact(l1);

  CGAL::Object inter=CGAL::intersection(t2,l2);
  const EK::Point_3& exact_pt=CGAL::object_cast<EK::Point_3>(inter);

  EK_to_IK to_inexact;

  IK::Point_3 inexact_pt = to_inexact(exact_pt);
  return 0;
}
```

File: examples/Kernel_23/cartesian_converter.cpp

See Also

CGAL::Cartesian<FieldNumberType> . . . . page 588
CGAL::Simple_cartesian<FieldNumberType> . . . . page 602

CGAL::cartesian_to_homogeneous

#include <CGAL/cartesian_homogeneous_conversion.h>

Point_2< Homogeneous<RT> > *cartesian_to_homogeneous(Point_2< Cartesian<RT> > cp)*

converts 2d point *cp* with Cartesian representation into a 2d point with homogeneous representation with the same number type.

Point_3< Homogeneous<RT> > *cartesian_to_homogeneous(Point_3< Cartesian<RT> > cp)*

converts 3d point *cp* with Cartesian representation into a 3d point with homogeneous representation with the same number type.

See Also

CGAL::Cartesian<FieldNumberType> ... page 588
CGAL::Cartesian_converter<K1, K2, NTConverter> ... page 589
CGAL::Homogeneous<RingNumberType> ... page 1070
CGAL::Homogeneous_converter<K1, K2, RTConverter, FTConverter> ... page 598
CGAL::homogeneous_to_cartesian ... page 599
CGAL::homogeneous_to_quotient_cartesian ... page 600
CGAL::quotient_cartesian_to_homogeneous ... page 764
CGAL::Simple_cartesian<FieldNumberType> ... page 602
CGAL::Simple_homogeneous<RingNumberType> ... page 603

Function

Class

CGAL::Filtered_kernel<CK>

Definition

*Filtered_kernel<CK>*is a kernel that uses the filtering technique based on interval arithmetic from [BBP01] to achieve exact and efficient predicates. In addition, a few selected important predicates are implemented using the formally proved, semi-static, filtering techniques from [MP05].

The geometric constructions are exactly those of the kernel *CK*, which means that they are not necessarily exact.

#include <CGAL/Filtered_kernel.h>

Parameters

The first parameter, *CK*, is the "Construction Kernel", namely the kernel from which are taken the types of the geometric objects as well as the geometric constructions.

The second parameter, *UseStaticFilters*, is a Boolean value which activates or not an additional layer of semi-static filters. It defaults to *true* (activated), unless the *CGAL_NO_STATIC_FILTERS* macro is defined. This option is mostly for debugging and testing, there should be no production use for deactivating static filters.

Is Model for the Concepts

Kernel

Types

typedef EK	*Exact_kernel;*	The type of the exact kernel.
typedef FK	*Approximate_kernel;*	The type of the approximate "filtering" kernel.

Constants

static const bool	*Has_static_filters;*	A Boolean value corresponding to the second template argument. Tells whether static filters are provided.

Example

The following example shows how to produce a kernel whose geometric objects and constructions are those of *Simple_cartesian<double>* but the predicates are exact.

```
#include <CGAL/Simple_cartesian.h>
#include <CGAL/Filtered_kernel.h>

typedef CGAL::Simple_cartesian<double> CK;
typedef CGAL::Filtered_kernel<CK> K;
```

Implementation

The implementation uses *CGAL::Filtered_predicate<EP, FP, C2E, C2F>* over each predicate of the kernel traits interface. Additionally, faster static filters may be used for a few selected critical predicates.

Class

CGAL::Filtered_kernel_adaptor<CK>

Definition

*Filtered_kernel_adaptor<CK>*is a kernel that uses the filtering technique [BBP01] to obtain a kernel with exact and efficient predicate functors. The geometric constructions are exactly those of the kernel *CK*, which means that they are not necessarily exact.

In contrast to *Filtered_kernel*, the global functions are those of *CK*.

Is Model for the Concepts

Kernel

#include <CGAL/Filtered_kernel.h>

Example

The following example shows how to produce a kernel whose geometric objects and constructions are those of *Simple_cartesian<double>* The predicate functors of the kernel are exact, the global functions are not.

```
#include <CGAL/Simple_cartesian.h>
#include <CGAL/Filtered_kernel.h>

typedef CGAL::Simple_cartesian<double> CK;
typedef CGAL::Filtered_kernel_adaptor<CK> K;


typedef K::Point_2 p(0,0), q(1,1), r(1,5);

CGAL::orientation(p,q,r);                // not exact

typedef K::Orientation_2 orientation;
orientation(p,q,r);                      // exact
```

CGAL::Filtered_predicate<EP, FP, C2E, C2F>

Class

Definition

Filtered_predicate<EP, FP, C2E, C2F> is an adaptor for predicate function objects that allows one to produce efficient and exact predicates. It is used to build *CGAL::Filtered_kernel<CK>* and can be used for other predicates too.

EP is the exact but supposedly slow predicate that is able to evaluate the predicate correctly. It will be called only when the filtering predicate, *FP*, cannot compute the correct result. This failure of *FP* must be done by throwing an exception.

To convert the geometric objects that are the arguments of the predicate, we use the function objects *C2E* and *C2F*, which must be of the form *Cartesian_converter* or *Homogeneous_converter*.

#include <CGAL/Filtered_predicate.h>

Types

typedef FP::result_type	*result_type;*	The return type of the function operators. It must also be the same type as *EP::result_type*.

Creation

Filtered_predicate<EP, FP, C2E, C2F> fo; — Default constructor.

Operations

template <class A1>
result_type *fo.operator()(A1 a1)* — The unary function operator for unary predicates.

template <class A1, class A2>
result_type *fo.operator()(A1 a1, A2 a2)*

The binary function operator for binary predicates.

Similar function operators are defined for up to 7 arguments.

Example

The following example defines an efficient and exact version of the orientation predicate over three points using the Cartesian representation with double coordinates and without reference counting (*Simple_cartesian< double>::Point_2*). Of course, the orientation predicate can already be found in the kernel, but you can follow this example to filter your own predicates. It uses the fast but inexact predicate based on interval arithmetic for filtering and the slow but exact predicate based on multi-precision floats when the filtering predicate fails.

```cpp
#include <CGAL/Simple_cartesian.h>
#include <CGAL/Filtered_predicate.h>
#include <CGAL/MP_Float.h>
#include <CGAL/Cartesian_converter.h>

typedef CGAL::Simple_cartesian<double> K;
typedef CGAL::Simple_cartesian<CGAL::Interval_nt_advanced> FK;
typedef CGAL::Simple_cartesian<CGAL::MP_Float> EK;
typedef CGAL::Cartesian_converter<K, EK> C2E;
typedef CGAL::Cartesian_converter<K, FK> C2F;

// Define my predicate, parameterized by a kernel.
template < typename K >
struct My_orientation_2
{
  typedef typename K::RT            RT;
  typedef typename K::Point_2       Point_2;

  typedef typename K::Orientation   result_type;

  result_type
  operator()(const Point_2 &p, const Point_2 &q, const Point_2 &r) const
  {
    RT prx = p.x() - r.x();
    RT pry = p.y() - r.y();
    RT qrx = q.x() - r.x();
    RT qry = q.y() - r.y();
    return CGAL::sign( prx*qry - qrx*pry );
  }
};

typedef CGAL::Filtered_predicate<My_orientation_2<EK>,
                                 My_orientation_2<FK>, C2E, C2F>
   Orientation_2;

int main()
{
  K::Point_2 p(1,2), q(2,3), r(3,4);
  Orientation_2 orientation;
  orientation(p, q, r);
  return 0;
}
```

File: examples/Filtered_kernel/Filtered_predicate.cpp

CGAL::Homogeneous<RingNumberType>

Class

#include <CGAL/Homogeneous.h>

Definition

A model for a *Kernel* using homogeneous coordinates to represent the geometric objects. In order for *Homogeneous<RingNumberType>* to model Euclidean geometry in E^2 and/or E^3, for some mathematical ring E (*e.g.*, the integers $\mathbb{Z}$ or the rationals $\mathbb{Q}$), the template parameter *RingNumberType* must model the mathematical ring E. That is, the ring operations on this number type must compute the mathematically correct results. If the number type provided as a model for *RingNumberType* is only an approximation of a ring (such as the built-in type *double*), then the geometry provided by the kernel is only an approximation of Euclidean geometry.

Is Model for the Concepts

Kernel . . . . page 581

Types

typedef Quotient<RingNumberType>	*FT;*
typedef RingNumberType	*RT;*

Implementation

This model of a kernel uses reference counting.

See Also

CGAL::Cartesian<FieldNumberType> . . . . page 588
CGAL::Simple_cartesian<FieldNumberType> . . . . page 602
CGAL::Simple_homogeneous<RingNumberType> . . . . page 603

CGAL::Homogeneous_converter<K1, K2, RTConverter, FTConverter>

Definition

*Homogeneous_converter<K1, K2, RTConverter, FTConverter>*converts objects from the kernel traits *K1* to the kernel traits *K2*. Those traits must be of the form *Homogeneous<RT1>* and *Homogeneous<RT2>* (or the equivalent with *Simple_homogeneous*). It then provides the following operators to convert objects from *K1* to *K2*.

The third template parameter *RT_Converter* is a function object that must provide *K2::RT operator()(const K1::RT &n);* that converts *n* to an *K2::RT* that has the same value.

The default value of this parameter is *CGAL::NT_converter<K1::RT, K2::RT>*, which uses the conversion operator from *K1::RT* to *K2::RT*.

Similarly, the fourth template parameter must provide *K2::FT operator()(const K1::FT &n);* that converts *n* to an *K2::FT* that has the same value. Its default value is *CGAL::NT_converter<K1::FT, K2::FT>*.

#include <CGAL/Homogeneous_converter.h>

Creation

Homogeneous_converter<K1, K2, RTConverter, FTConverter> conv;

Default constructor.

Operations

K2::Point_2 *conv.operator()(K1::Point_2 p)*

returns a *K2::Point_2* which coordinates are those of *p*, converted by *RTConverter*.

Similar operators are defined for the other kernel traits geometric types *Point_3*, *Vector_2*...

See Also

CGAL::Homogeneous<RingNumberType> . . . . page 1070
CGAL::Simple_homogeneous<RingNumberType> . . . . page 603

CGAL::homogeneous_to_cartesian

#include <CGAL/cartesian_homogeneous_conversion.h>

Point_2< Cartesian<FT> > *homogeneous_to_cartesian(Point_2< Homogeneous<FT> > hp)*

converts 2d point *hp* with homogeneous representation into a 2d point with Cartesian representation with the same number type.

Point_3< Cartesian<FT> > *homogeneous_to_cartesian(Point_3< Homogeneous<FT> > hp)*

converts 3d point *hp* with homogeneous representation into a 3d point with Cartesian representation with the same number type.

See Also

See Also

CGAL::Cartesian<FieldNumberType> . . . . page 588
CGAL::Cartesian_converter<K1, K2, NTConverter> . . . . page 589
CGAL::cartesian_to_homogeneous . . . . page 591
CGAL::Homogeneous<RingNumberType> . . . . page 1070
CGAL::Homogeneous_converter<K1, K2, RTConverter, FTConverter> . . . . page 598
CGAL::homogeneous_to_quotient_cartesian . . . . page 600
CGAL::quotient_cartesian_to_homogeneous . . . . page 764
CGAL::Simple_cartesian<FieldNumberType> . . . . page 602
CGAL::Simple_homogeneous<RingNumberType> . . . . page 603

Function

Function

CGAL::homogeneous_to_quotient_cartesian

#include <CGAL/cartesian_homogeneous_conversion.h>

Point_2< Cartesian<Quotient<RT> > >

homogeneous_to_quotient_cartesian(Point_2<Homogeneous<RT> > hp)

converts the 2d point *hp* with homogeneous representation with number type *RT* into a 2d point with Cartesian representation with number type *Quotient<RT>*.

Point_3< Cartesian<Quotient<RT> > >

homogeneous_to_quotient_cartesian(Point_3<Homogeneous<RT> > hp)

converts the 3d point *hp* with homogeneous representation with number type *RT* into a 3d point with Cartesian representation with number type *Quotient<RT>*.

See Also

CGAL::Cartesian<FieldNumberType> . . . . . . page 588
CGAL::Cartesian_converter<K1, K2, NTConverter> . . . . . . page 589
CGAL::cartesian_to_homogeneous . . . . . . page 591
CGAL::Homogeneous<RingNumberType> . . . . . . page 1070
CGAL::Homogeneous_converter<K1, K2, RTConverter, FTConverter> . . . . . . page 598
CGAL::homogeneous_to_cartesian . . . . . . page 599
CGAL::quotient_cartesian_to_homogeneous . . . . . . page 764
CGAL::Simple_cartesian<FieldNumberType> . . . . . . page 602
CGAL::Simple_homogeneous<RingNumberType> . . . . . . page 603

CGAL::Kernel_traits<T>

#include <CGAL/Kernel_traits.h>

Class

Definition

The class *Kernel_traits<T>* provides access to the kernel model to which the argument type *T* belongs. (Provided *T* belongs to some kernel model.) The default implementation assumes there is a local type *T::Kernel* referring to the kernel model of *T*. If this type does not exist, a specialization of *Kernel_traits<T>* can be used to provide the desired information.

This class is, for example, useful in the following context. Assume you want to write a generic function that accepts two points p and q as argument and constructs the line segment between p and q. In order to specify the return type of this function, you need to know what is the segment type corresponding to the Point type representing p and q. Using *Kernel_traits<T>*, this can be done as follows.

```
template < class Point >
typename Kernel_traits<Point>::Kernel::Segment
construct_segment(Point p, Point q)
{ ... }
```

Types

typedef T::R	*Kernel;*	If *T* is a type *K::Point_2* of some kernel model *K*, then *Kernel* is equal to *K*.

Class

CGAL::Simple_cartesian<FieldNumberType>

#include <CGAL/Simple_cartesian.h>

Definition

A model for a *Kernel* using Cartesian coordinates to represent the geometric objects. In order for *Simple_cartesian<FieldNumberType>* to model Euclidean geometry in E^2 and/or E^3, for some mathematical field E (*e.g.*, the rationals $\mathbb{Q}$ or the reals $\mathbb{R}$), the template parameter *FieldNumberType* must model the mathematical field E. That is, the field operations on this number type must compute the mathematically correct results. If the number type provided as a model for *FieldNumberType* is only an approximation of a field (such as the built-in type *double*), then the geometry provided by the kernel is only an approximation of Euclidean geometry.

Is Model for the Concepts

Kernel . . . . page 581

Types

typedef FieldNumberType *FT;*

typedef FieldNumberType *RT;*

Implementation

In contrast to *Cartesian*, no reference counting is used internally. This eases debugging, but may slow down algorithms that copy objects intensively.

See Also

CGAL::Cartesian<FieldNumberType> . . . . page 588
CGAL::Homogeneous<RingNumberType> . . . . page 1070
CGAL::Simple_homogeneous<RingNumberType> . . . . page 603

CGAL::Simple_homogeneous<RingNumberType>

#include <CGAL/Simple_homogeneous.h>

Class

Definition

A model for a *Kernel* using homogeneous coordinates to represent the geometric objects. In order for *Simple_homogeneous<RingNumberType>* to model Euclidean geometry in E^2 and/or E^3, for some mathematical ring E (*e.g.*, the integers $\mathbb{Z}$ or the rationals $\mathbb{Q}$), the template parameter *RingNumberType* must model the mathematical ring E. That is, the ring operations on this number type must compute the mathematically correct results. If the number type provided as a model for *RingNumberType* is only an approximation of a ring (such as the built-in type *double*), then the geometry provided by the kernel is only an approximation of Euclidean geometry.

Is Model for the Concepts

Kernel . . . . page 581

Types

typedef Quotient<RingNumberType>	*FT;*
typedef RingNumberType	*RT;*

Implementation

In contrast to *Homogeneous*, no reference counting is used internally. This eases debugging, but may slow down algorithms that copy objects intensively, or slightly speed up others.

See Also

CGAL::Cartesian<FieldNumberType> . . . . page 588
CGAL::Homogeneous<RingNumberType> . . . . page 1070
CGAL::Simple_cartesian<FieldNumberType> . . . . page 602

Class

CGAL::Projection_traits_xy_3<K>

Definition

The class *Projection_traits_xy_3<K>* is an adapter to apply 2D algorithms to the projections of 3D data on the *xy*-plane.

CGAL provides also predefined geometric traits classes *Projection_traits_yz_3<K>* and *Projection_traits_xz_3<K>* to deal with projections on the *zx*- and the *zy*-plane, respectively.

Parameters

The template parameter *K* has to be instantiated by a model of the *Kernel* concept. *Projection_traits_xy_3<K>* uses types and predicates defined in *K*.

#include <CGAL/Projection_traits_xy_3.h>

Is Model for the Concepts

The class is a model of several 2D triangulation traits class concepts, except that it does not provide the type and constructors required to build the dual Voronoi diagram. The class is also a model of the concepts *PolygonTraits_2* and *ConvexHullTraits_2*.

TriangulationTraits_2 .... page 2541
DelaunayTriangulationTraits_2 .... page 2513
ConstrainedTriangulationTraits_2 .... page 2495
PolygonTraits_2 .... page 1470
ConvexHullTraits_2 .... page 1381

Types

typedef Point_3<K>	*Point_2;*
typedef Segment_3<K>	*Segment_2;*
typedef Triangle_3<K>	*Triangle_2;*
typedef Line_3<K>	*Line_2;*

typedef K::Object_3	*Object_2;*

The functors provided by this class are those listed in the concepts, except that it does not provide the type and constructors required to build the dual Voronoi diagram. The functors operate on the 2D projection of their arguments. They come with preconditions that projections of the arguments are non-degenerate, eg. a line segment does not project on a single point, two points do not project on the same point, etc.

In the following, we specify the choice of the *z*-coordinate in case a new point is constructed.

Projection_traits_xy_3<K>:: Intersect_2

A construction object. Provides the operator :
Object_2 operator()(Segment_2 s1, Segment_2 s2); which returns a 3D object whose projection on the xy-plane is the intersection of the projections of *s1* and *s2*. If non empty, the returned object is either a segment or a point. Its embedding in 3D is computed as the interpolation between *s1* and *s2*, meaning that any point *p* of the returned object is the midpoint of segment *p1p2* where *p1* and *p2* are the two points of *s1* and *s2* respectively, both projecting on *p*.
Precondition: The projection of *s1* and the projection of *s2* are non-degenerate *2D* segments.

Creation

Projection_traits_xy_3<K> traits;

default constructor.

Projection_traits_xy_3<K> traits(Projection_traits_xy_3 tr);

Copy constructor.

Projection_traits_xy_3 *traits.operator=(Projection_traits_xy_3 tr)*

Assignment operator.

11.10 Predefined Kernels

Class

CGAL::Exact_predicates_exact_constructions_kernel

Definition

A typedef to a kernel which has the following properties:

- It uses Cartesian representation.
- It supports constructions of points from `double` Cartesian coordinates.
- It provides both exact geometric predicates and exact geometric constructions.

#include <CGAL/Exact_predicates_exact_constructions_kernel.h>

Is Model for the Concepts

Kernel

See Also

CGAL::Exact_predicates_exact_constructions_kernel_with_sqrt
CGAL::Exact_predicates_inexact_constructions_kernel
CGAL::Cartesian

CGAL::Exact_predicates_exact_constructions_kernel_with_sqrt

Class

Definition

A typedef to a kernel which has the following properties:

- It uses Cartesian representation.
- It supports constructions of points from `double` Cartesian coordinates.
- It provides both exact geometric predicates and exact geometric constructions.
- Its *FT* nested type supports the square root operation *sqrt()*.

Note that it requires CORE or LEDA installed.

#include <CGAL/Exact_predicates_exact_constructions_kernel_with_sqrt.h>

Is Model for the Concepts

Kernel

See Also

CGAL::Exact_predicates_exact_constructions_kernel
CGAL::Exact_predicates_inexact_constructions_kernel
CGAL::Cartesian

Class

CGAL::Exact_predicates_inexact_constructions_kernel

Definition

A typedef to a kernel which has the following properties:

- It uses Cartesian representation.
- It supports constructions of points from `double` Cartesian coordinates.
- It provides exact geometric predicates, but inexact geometric constructions.

#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>

Is Model for the Concepts

Kernel

See Also

CGAL::Exact_predicates_exact_constructions_kernel
CGAL::Exact_predicates_exact_constructions_kernel_with_sqrt
CGAL::Cartesian

11.11 Kernel Objects

11.11.1 Two-dimensional Objects

CGAL::Aff_transformation_2<Kernel>

Definition

The class *Aff_transformation_2<Kernel>* represents two-dimensional affine transformations. The general form of an affine transformation is based on a homogeneous representation of points. Thereby all transformations can be realized by matrix multiplications.

Multiplying the transformation matrix by a scalar does not change the represented transformation. Therefore, any transformation represented by a matrix with rational entries can be represented by a transformation matrix with integer entries as well. (Multiply the matrix with the common denominator of the rational entries.) Hence, it is sufficient to use the number type *Kernel::RT* to represent the entries of the transformation matrix.

CGAL offers several specialized affine transformations. Different constructors are provided to create them. They are parameterized with a symbolic name to denote the transformation type, followed by additional parameters. The symbolic name tags solve ambiguities in the function overloading and they make the code more readable, i.e., what type of transformation is created.

Since two-dimensional points have three homogeneous coordinates, we have a 3×3 matrix $(m_{ij})_{i,\, j=0\ldots 2}$.

If the homogeneous representations are normalized (the homogenizing coordinate is 1), then the upper left 2×2 matrix realizes linear transformations. In the matrix form of a translation, the translation vector $(v_0,\, v_1,\, 1)$ appears in the last column of the matrix. The entries m_{20} and m_{21} are always zero and therefore do not appear in the constructors.

Creation

Aff_transformation_2<Kernel> t(Identity_transformation);

introduces an identity transformation.

Aff_transformation_2<Kernel> t(const Translation, Vector_2<Kernel> v);

introduces a translation by a vector *v*.

Aff_transformation_2<Kernel> t(const Rotation,
Direction_2<Kernel> d,
Kernel::RT num,

Kernel::RT den = RT(1))

approximates the rotation over the angle indicated by direction d, such that the differences between the sines and cosines of the rotation given by d and the approximating rotation are at most num/den each.
Precondition: $num/den > 0$ and $d! = 0$.

Aff_transformation_2<Kernel> t(const Rotation, Kernel::RT sine_rho, Kernel::RT cosine_rho, Kernel::RT hw = RT(1))

introduces a rotation by the angle *rho*.
Precondition: $sine\_rho^2 + cosine\_rho^2 == hw^2$.

Aff_transformation_2<Kernel> t(const Scaling, Kernel::RT s, Kernel::RT hw = RT(1));

introduces a scaling by a scale factor s/hw.

Aff_transformation_2<Kernel> t(Kernel::RT m00, Kernel::RT m01, Kernel::RT m02, Kernel::RT m10, Kernel::RT m11, Kernel::RT m12, Kernel::RT hw = RT(1))

introduces a general affine transformation in the 3×3 matrix form $\begin{pmatrix} m_{00} & m_{01} & m_{02} \\ m_{10} & m_{11} & m_{12} \\ 0 & 0 & hw \end{pmatrix}$. The sub-matrix $\frac{1}{hw}\begin{pmatrix} m_{00} & m_{01} \\ m_{10} & m_{11} \end{pmatrix}$ contains the scaling and rotation information, the vector $\frac{1}{hw}\begin{pmatrix} m_{02} \\ m_{12} \end{pmatrix}$ contains the translational part of the transformation.

Aff_transformation_2<Kernel> t(Kernel::RT m00, Kernel::RT m01, Kernel::RT m10, Kernel::RT m11, Kernel::RT hw = RT(1))

introduces a general linear transformation $\begin{pmatrix} m_{00} & m_{01} & 0 \\ m_{10} & m_{11} & 0 \\ 0 & 0 & hw \end{pmatrix}$, i.e. there is no translational part.

Operations

The main thing to do with transformations is to apply them on geometric objects. Each class *Class_2<Kernel>* representing a geometric object has a member function:

Class_2<Kernel> transform(Aff_transformation_2<Kernel> t).

The transformation classes provide a member function *transform()* for points, vectors, directions, and lines:

Point_2<Kernel>	*t.transform(Point_2<Kernel> p) const*
Vector_2<Kernel>	*t.transform(Vector_2<Kernel> p) const*
Direction_2<Kernel>	*t.transform(Direction_2<Kernel> p) const*
Line_2<Kernel>	*t.transform(Line_2<Kernel> p) const*

CGAL provides function operators for these member functions:

Point_2<Kernel>	*t.operator()(Point_2<Kernel> p) const*
Vector_2<Kernel>	*t.operator()(Vector_2<Kernel> p) const*
Direction_2<Kernel>	*t.operator()(Direction_2<Kernel> p) const*
Line_2<Kernel>	*t.operator()(Line_2<Kernel> p) const*

Miscellaneous

Aff_transformation_2<Kernel>		
	t.operator(s) const*	composes two affine transformations.
Aff_transformation_2<Kernel>		
	t.inverse() const	gives the inverse transformation.
bool	*t.is_even() const*	returns *true*, if the transformation is not reflecting, i.e. the determinant of the involved linear transformation is non-negative.
bool	*t.is_odd() const*	returns *true*, if the transformation is reflecting.

The matrix entries of a matrix representation of a *Aff_transformation_2<Kernel>* can be accessed trough the following member functions:

Kernel::FT	*t.cartesian(int i, int j) const*	
Kernel::FT	*t.m(int i, int j) const*	returns entry m_{ij} in a matrix representation in which m_{22} is 1.
Kernel::RT	*t.homogeneous(int i, int j) const*	
Kernel::RT	*t.hm(int i, int j) const*	returns entry m_{ij} in some fixed matrix representation.

For affine transformations no I/O operators are defined.

See Also

Identity_transformation, *Rotation*, *Scaling*, *Translation*
rational_rotation_approximation

Example

```
typedef Cartesian<double>        K;
typedef Aff_transformation_2<K>  Transformation;
typedef Point_2<K>               Point;
typedef Vector_2<K>              Vector;
typedef Direction_2<K>           Direction;

Transformation rotate(ROTATION, sin(pi), cos(pi));
Transformation rational_rotate(ROTATION,Direction(1,1), 1, 100);
Transformation translate(TRANSLATION, Vector(-2, 0));
Transformation scale(SCALING, 3);

Point q(0, 1);
q = rational_rotate(q);

Point p(1, 1);

p = rotate(p);

p = translate(p);

p = scale(p);
```

The same would have been achieved with

```
Transformation transform = scale * (translate * rotate);
p = transform(Point(1.0, 1.0));
```

See Also

CGAL::Bbox_2

#include <CGAL/Bbox_2.h>

Definition

An object *b* of the class *Bbox_2* is a bounding box in the two-dimensional Euclidean plane $\mathbb{E}^2$. This class is not templated.

Creation

Bbox_2 b(double x_min, double y_min, double x_max, double y_max);

introduces a bounding box *b* with lower left corner at *(xmin, ymin)* and with upper right corner at *(xmax, ymax)*.

Operations

bool	*b.operator==(c) const*	
		Test for equality.
bool	*b.operator!=(q) const*	
		Test for inequality.
int	*b.dimension() const*	Returns 2.
double	*b.xmin() const*	
double	*b.ymin() const*	
double	*b.xmax() const*	
double	*b.ymax() const*	
double	*b.min(int i) const*	Returns *xmin()* if *i==0* or *ymin()* if *i==1*. *Precondition*: i==0 or i==1
double	*b.max(int i) const*	Returns *xmax()* if *i==0* or *ymax()* if *i==1*. *Precondition*: i==0 or i==1
Bbox_2	*b.operator+(c) const*	returns a bounding box of *b* and *c*.

See Also

CGAL::Bbox_3 . . . . page 639
CGAL::do_overlap . . . . page 732

Class

CGAL::Circle_2<Kernel>

Definition

An object of type *Circle_2<Kernel>* is a circle in the two-dimensional Euclidean plane $\mathbb{E}^2$. The circle is oriented, i.e. its boundary has clockwise or counterclockwise orientation. The boundary splits $\mathbb{E}^2$ into a positive and a negative side, where the positive side is to the left of the boundary. The boundary also splits $\mathbb{E}^2$ into a bounded and an unbounded side. Note that the circle can be degenerated, i.e. the squared radius may be zero.

Creation

Circle_2<Kernel> c(Point_2<Kernel> center, Kernel::FT squared_radius, Orientation ori = COUNTERCLOCKWISE)

introduces a variable c of type *Circle_2<Kernel>*. It is initialized to the circle with center *center*, squared radius *squared_radius* and orientation *ori*.
Precondition: $ori \neq COLLINEAR$, and further, $squared\_radius \geq 0$.

Circle_2<Kernel> c(Point_2<Kernel> p, Point_2<Kernel> q, Point_2<Kernel> r);

introduces a variable c of type *Circle_2<Kernel>*. It is initialized to the unique circle which passes through the points p, q and r. The orientation of the circle is the orientation of the point triple p, q, r.
Precondition: p, q, and r are not collinear.

Circle_2<Kernel> c(Point_2<Kernel> p, Point_2<Kernel> q, Orientation ori = COUNTERCLOCKWISE);

introduces a variable c of type *Circle_2<Kernel>*. It is initialized to the circle with diameter $\overline{pq}$ and orientation *ori*.
Precondition: $ori \neq COLLINEAR$.

Circle_2<Kernel> c(Point_2<Kernel> center, Orientation ori = COUNTERCLOCKWISE);

introduces a variable c of type *Circle_2<Kernel>*. It is initialized to the circle with center *center*, squared radius zero and orientation *ori*.
Precondition: $ori \neq COLLINEAR$.
Postcondition: $c.is\_degenerate() = true$.

Access Functions

Point_2<Kernel>	*c.center() const*	returns the center of c.

Kernel::FT	*c.squared_radius() const*	
		returns the squared radius of *c*.
Orientation	*c.orientation() const*	returns the orientation of *c*.
bool operator	*c.==(circle2) const*	returns *true*, iff *c* and *circle2* are equal, i.e. if they have the same center, same squared radius and same orientation.
bool operator	*c.!=(circle2) const*	returns *true*, iff *c* and *circle2* are not equal.

Predicates

bool	*c.is_degenerate() const*	
		returns *true*, iff *c* is degenerate, i.e. if *c* has squared radius zero.
Oriented_side	*c.oriented_side(Point_2<Kernel> p) const*	
		returns either the constant *ON_ORIENTED_BOUNDARY*, *ON_POSITIVE_SIDE*, or *ON_NEGATIVE_SIDE*, iff *p* lies on the boundary, properly on the positive side, or properly on the negative side of *c*, resp.
Bounded_side	*c.bounded_side(Point_2<Kernel> p) const*	
		returns *ON_BOUNDED_SIDE*, *ON_BOUNDARY*, or *ON_UNBOUNDED_SIDE* iff *p* lies properly inside, on the boundary, or properly outside of *c*, resp.
bool	*c.has_on_positive_side(Point_2<Kernel> p) const*	
bool	*c.has_on_negative_side(Point_2<Kernel> p) const*	
bool	*c.has_on_boundary(Point_2<Kernel> p) const*	
bool	*c.has_on_bounded_side(Point_2<Kernel> p) const*	
bool	*c.has_on_unbounded_side(Point_2<Kernel> p) const*	

Miscellaneous

Circle_2<Kernel>	*c.opposite() const*	returns the circle with the same center and squared radius as *c* but with opposite orientation.
Circle_2<Kernel>	*c.orthogonal_transform(Aff_transformation_2<Kernel> at) const*	
		returns the circle obtained by applying *at* on *c*. *Precondition*: *at* is an orthogonal transformation.

Bbox_2	*c.bbox() const*	returns a bounding box containing c.

See Also

Kernel::Circle_2 . . . . page 783

CGAL::Direction_2<Kernel>

Class

Definition

An object of the class *Direction_2<Kernel>* is a vector in the two-dimensional vector space $\mathbb{R}^2$ where we forget about its length. They can be viewed as unit vectors, although there is no normalization internally, since this is error prone. Directions are used whenever the length of a vector does not matter. They also characterize a set of parallel oriented lines that have the same orientations. For example, you can ask for the direction orthogonal to an oriented plane, or the direction of an oriented line. Further, they can be used to indicate angles. The slope of a direction is *dy()/dx()*.

Creation

Direction_2<Kernel> d(Vector_2<Kernel> v); introduces the direction d of vector v.

Direction_2<Kernel> d(Line_2<Kernel> l); introduces the direction d of line l.

Direction_2<Kernel> d(Ray_2<Kernel> r); introduces the direction d of ray r.

Direction_2<Kernel> d(Segment_2<Kernel> s); introduces the direction d of segment s.

Direction_2<Kernel> d(Kernel::RT x, Kernel::RT y);

introduces a direction d passing through the origin and the point with Cartesian coordinates (x,y).

Operations

Kernel::RT	*d.delta(int i) const*	returns values, such that *d== Direction_2<Kernel> (delta(0),delta(1))*. *Precondition*: : $0 \leq i \leq 1$.
Kernel::RT	*d.dx() const*	returns *delta(0)*.
Kernel::RT	*d.dy() const*	returns *delta(1)*.

There is a total order on directions. We compare the angles between the positive x-axis and the directions in counterclockwise order.

bool	*d.operator==(e) const*
bool	*d.operator!=(e) const*
bool	*d.operator<(e) const*
bool	*d.operator>(e) const*
bool	*d.operator<=(e) const*
bool	*d.operator>=(e) const*

Furthermore, we have

bool	*d.counterclockwise_in_between(d1, d2) const*	
		returns true, iff *d* is not equal to *d1*, and while rotating counterclockwise starting at *d1*, *d* is reached strictly before *d2* is reached. Note that true is returned if *d1* == *d2*, unless also *d* == *d1*.
Direction_2<Kernel>	*d.operator-() const*	The direction opposite to *d*.

Miscellaneous

Vector_2<Kernel>	*d.vector() const*	returns a vector that has the same direction as *d*.
Direction_2<Kernel>	*d.transform(Aff_transformation_2<Kernel> t) const*	
		returns the direction obtained by applying *t* on *d*.

See Also

Kernel::Direction_2 . . . . page 785

CGAL::Iso_rectangle_2<Kernel>

Class

Definition

An object s of the data type *Iso_rectangle_2<Kernel>* is a rectangle in the Euclidean plane $\mathbb{E}^2$ with sides parallel to the x and y axis of the coordinate system.

Although they are represented in a canonical form by only two vertices, namely the lower left and the upper right vertex, we provide functions for "accessing" the other vertices as well. The vertices are returned in counterclockwise order.

Iso-oriented rectangles and bounding boxes are quite similar. The difference however is that bounding boxes have always double coordinates, whereas the coordinate type of an iso-oriented rectangle is chosen by the user.

Creation

Iso_rectangle_2<Kernel> r(Point_2<Kernel> p, Point_2<Kernel> q);

introduces an iso-oriented rectangle r with diagonal opposite vertices p and q. Note that the object is brought in the canonical form.

Iso_rectangle_2<Kernel> r(Point_2<Kernel> p, Point_2<Kernel> q, int);

introduces an iso-oriented rectangle r with diagonal opposite vertices p and q. The *int* argument value is only used to distinguish the two overloaded functions.
Precondition: $p.x() <= q.x()$ and $p.y() <= q.y()$.

Iso_rectangle_2<Kernel> r(Point_2<Kernel> left,
Point_2<Kernel> right,
Point_2<Kernel> bottom,
Point_2<Kernel> top)

introduces an iso-oriented rectangle r whose minimal x coordinate is the one of *left*, the maximal x coordinate is the one of *right*, the minimal y coordinate is the one of *bottom*, the maximal y coordinate is the one of *top*.

Iso_rectangle_2<Kernel> r(Kernel::RT min_hx,
Kernel::RT min_hy,
Kernel::RT max_hx,
Kernel::RT max_hy,
Kernel::RT hw = RT(1))

introduces an iso-oriented rectangle r with diagonal opposite vertices (*min_hx/hw*, *min_hy/hw*) and (*max_hx/hw*, *max_hy/hw*).
Precondition: $hw \neq 0$.

Operations

bool	*r.operator==(r2) const*	
		Test for equality: two iso-oriented rectangles are equal, iff their lower left and their upper right vertices are equal.
bool	*r.operator!=(r2) const*	
		Test for inequality.
Point_2<Kernel>	*r.vertex(int i) const*	returns the i'th vertex modulo 4 of *r* in counterclockwise order, starting with the lower left vertex.
Point_2<Kernel>	*r.operator[](int i) const*	
		returns *vertex(i)*.
Point_2<Kernel>	*r.min() const*	returns the lower left vertex of *r* (= *vertex(0)*).
Point_2<Kernel>	*r.max() const*	returns the upper right vertex of *r* (= *vertex(2)*).
Kernel::FT	*r.xmin() const*	returns the x coordinate of lower left vertex of *r*.
Kernel::FT	*r.ymin() const*	returns the y coordinate of lower left vertex of *r*.
Kernel::FT	*r.xmax() const*	returns the x coordinate of upper right vertex of *r*.
Kernel::FT	*r.ymax() const*	returns the y coordinate of upper right vertex of *r*.
Kernel::FT	*r.min_coord(int i) const*	
		returns the i'th Cartesian coordinate of the lower left vertex of *r*. *Precondition*: $0 \leq i \leq 1$.
Kernel::FT	*r.max_coord(int i) const*	
		returns the i'th Cartesian coordinate of the upper right vertex of *r*. *Precondition*: $0 \leq i \leq 1$.

Predicates

bool	*r.is_degenerate() const*	
		r is degenerate, if all vertices are collinear.

Bounded_side	*r.bounded_side(Point_2<Kernel> p) const*	
		returns either *ON_UNBOUNDED_SIDE*, *ON_BOUNDED_SIDE*, or the constant *ON_BOUNDARY*, depending on where point p is.
bool	*r.has_on_boundary(Point_2<Kernel> p) const*	
bool	*r.has_on_bounded_side(Point_2<Kernel> p) const*	
bool	*r.has_on_unbounded_side(Point_2<Kernel> p) const*	

Miscellaneous

Kernel::FT	*r.area() const*	returns the area of r.
Bbox	*r.bbox() const*	returns a bounding box containing r.
Iso_rectangle_2<Kernel>		
	r.transform(Aff_transformation_2<Kernel> t) const	
		returns the iso-oriented rectangle obtained by applying t on the lower left and the upper right corner of r. *Precondition*: The angle at a rotation must be a multiple of $\pi/2$, otherwise the resulting rectangle does not have the same side length. Note that rotating about an arbitrary angle can even result in a degenerate iso-oriented rectangle.

See Also

Kernel::IsoRectangle_2 . . . . page 788

Class

CGAL::Line_2<Kernel>

Definition

An object l of the data type *Line_2<Kernel>* is a directed straight line in the two-dimensional Euclidean plane $\mathbb{E}^2$. It is defined by the set of points with Cartesian coordinates (x,y) that satisfy the equation

$$l:\ ax+by+c=0.$$

The line splits $\mathbb{E}^2$ in a *positive* and a *negative* side. A point p with Cartesian coordinates (px,py) is on the positive side of l, iff $a\,px+b\,py+c>0$, it is on the negative side of l, iff $a\,px+b\,py+c<0$. The positive side is to the left of l.

Creation

Line_2<Kernel> l(Kernel::RT a, Kernel::RT b, Kernel::RT c);

introduces a line l with the line equation in Cartesian coordinates $ax+by+c=0$.

Line_2<Kernel> l(Point_2<Kernel> p, Point_2<Kernel> q);

introduces a line l passing through the points p and q. Line l is directed from p to q.

Line_2<Kernel> l(Point_2<Kernel> p, Direction_2<Kernel> d);

introduces a line l passing through point p with direction d.

Line_2<Kernel> l(Point_2<Kernel> p, Vector_2<Kernel> v);

introduces a line l passing through point p and oriented by v.

Line_2<Kernel> l(Segment_2<Kernel> s); introduces a line l supporting the segment s, oriented from source to target.

Line_2<Kernel> l(Ray_2<Kernel> r); introduces a line l supporting the ray r, with same orientation.

Operations

bool	*l.operator==(h) const*	Test for equality: two lines are equal, iff they have a non empty intersection and the same direction.

bool	*l.operator!=(h) const*	Test for inequality.
Kernel::RT	*l.a() const*	returns the first coefficient of l.
Kernel::RT	*l.b() const*	returns the second coefficient of l.
Kernel::RT	*l.c() const*	returns the third coefficient of l.
Point_2<Kernel>	*l.point(int i) const*	returns an arbitrary point on *l*. It holds *point(i) == point(j)*, iff *i==j*. Furthermore, *l* is directed from *point(i)* to *point(j)*, for all $i < j$.
Point_2<Kernel>	*l.projection(Point_2<Kernel> p) const*	returns the orthogonal projection of *p* onto *l*.
Kernel::FT	*l.x_at_y(Kernel::FT y) const*	returns the *x*-coordinate of the point at *l* with given *y*-coordinate. *Precondition*: *l* is not horizontal.
Kernel::FT	*l.y_at_x(Kernel::FT x) const*	returns the *y*-coordinate of the point at *l* with given *x*-coordinate. *Precondition*: *l* is not vertical.

Predicates

bool	*l.is_degenerate() const*	line *l* is degenerate, if the coefficients *a* and *b* of the line equation are zero.
bool	*l.is_horizontal() const*	
bool	*l.is_vertical() const*	
Oriented_side	*l.oriented_side(Point_2<Kernel> p) const*	returns *ON_ORIENTED_BOUNDARY*, *ON_NEGATIVE_SIDE*, or the constant *ON_POSITIVE_SIDE*, depending on the position of *p* relative to the oriented line *l*.

For convenience we provide the following Boolean functions:

bool	*l.has_on(Point_2<Kernel> p) const*
bool	*l.has_on_positive_side(Point_2<Kernel> p) const*
bool	*l.has_on_negative_side(Point_2<Kernel> p) const*

Miscellaneous

Vector_2<Kernel>	*l.to_vector() const*	returns a vector having the direction of *l*.
Direction_2<Kernel>	*l.direction() const*	returns the direction of *l*.
Line_2<Kernel>	*l.opposite() const*	returns the line with opposite direction.
Line_2<Kernel>	*l.perpendicular(Point_2<Kernel> p) const*	returns the line perpendicular to *l* and passing through *p*, where the direction is the direction of *l* rotated counterclockwise by 90 degrees.
Line_2<Kernel>	*l.transform(Aff_transformation_2<Kernel> t) const*	returns the line obtained by applying *t* on a point on *l* and the direction of *l*.

Example

Let us first define two Cartesian two-dimensional points in the Euclidean plane $\mathbb{E}^2$. Their dimension and the fact that they are Cartesian is expressed by the suffix *_2* and the representation type *Cartesian*.

```
Point_2< Cartesian<double> >  p(1.0,1.0), q(4.0,7.0);
```

To define a line *l* we write:

```
Line_2< Cartesian<double> > l(p,q);
```

See Also

Kernel::Line_2 . . . . page 789

CGAL::Point_2<Kernel>

Class

Definition

An object of the class *Point_2<Kernel>* is a point in the two-dimensional Euclidean plane $\mathbb{E}^2$.

Remember that *Kernel::RT* and *Kernel::FT* denote a *RingNumberType* and a *FieldNumberType*, respectively. For the kernel model *Cartesian<T>*, the two types are the same. For the kernel model *Homogeneous<T>*, *Kernel::RT* is equal to *T*, and *Kernel::FT* is equal to *Quotient<T>*.

Types

Point_2<Kernel>:: Cartesian_const_iterator

An iterator for enumerating the Cartesian coordinates of a point.

Creation

Point_2<Kernel> p(Origin ORIGIN);

introduces a variable p with Cartesian coordinates $(0,0)$.

Point_2<Kernel> p(int x, int y);

introduces a point p initialized to (x,y).

Point_2<Kernel> p(double x, double y);

introduces a point p initialized to (x,y) provided *RT* supports construction from *double*.

Point_2<Kernel> p(Kernel::RT hx, Kernel::RT hy, Kernel::RT hw = RT(1));

introduces a point p initialized to $(hx/hw, hy/hw)$.
Precondition: $hw \neq$ *Kernel::RT(0)*

Point_2<Kernel> p(Kernel::FT x, Kernel::FT y);

introduces a point p initialized to (x,y).

Operations

bool *p.operator==(q) const*

Test for equality. Two points are equal, iff their x and y coordinates are equal. The point can be compared with *ORIGIN*.

bool *p.operator!=(q) const*

Test for inequality. The point can be compared with *ORIGIN*.

There are two sets of coordinate access functions, namely to the homogeneous and to the Cartesian coordinates. They can be used independently from the chosen kernel model.

Kernel::RT *p.hx() const*

returns the homogeneous x coordinate.

Kernel::RT *p.hy() const*

returns the homogeneous y coordinate.

Kernel::RT *p.hw() const*

returns the homogenizing coordinate.

Note that you do not loose information with the homogeneous representation, because the *FieldNumberType* is a quotient.

Kernel::FT *p.x() const*

returns the Cartesian x coordinate, that is hx/hw.

Kernel::FT *p.y() const*

returns the Cartesian y coordinate, that is hy/hw.

The following operations are for convenience and for compatibility with higher dimensional points. Again they come in a Cartesian and in a homogeneous flavor.

Kernel::RT *p.homogeneous(int i) const*

returns the i'th homogeneous coordinate of p, starting with 0.
Precondition: $0 \leq i \leq 2$.

Kernel::FT *p.cartesian(int i) const*

returns the i'th Cartesian coordinate of p, starting with 0.
Precondition: $0 \leq i \leq 1$.

Kernel::FT *p.operator[](int i) const*

returns *cartesian(i)*.
Precondition: $0 \leq i \leq 1$.

Cartesian_const_iterator *p.cartesian_begin() const*

returns an iterator to the Cartesian coordinates of p, starting with the 0th coordinate.

Cartesian_const_iterator *p.cartesian_end() const*

returns an off the end iterator to the Cartesian coordinates of p.

int *p.dimension() const*

returns the dimension (the constant 2).

Bbox_2 *p.bbox() const*

returns a bounding box containing p. Note that bounding boxes are not parameterized with whatsoever.

Point_2<Kernel> *p.transform(Aff_transformation_2<Kernel> t) const*

returns the point obtained by applying t on p.

Operators

The following operations can be applied on points:

bool *operator<(p, q)*

returns true iff p is lexicographically smaller than q, i.e. either if $p.x() < q.x()$ or if $p.x() == q.x()$ and $p.y() < q.y()$.

bool *operator>(p, q)*

returns true iff p is lexicographically greater than q.

bool *operator<=(p, q)*

returns true iff p is lexicographically smaller or equal to q.

bool *operator>=(p, q)*

returns true iff p is lexicographically greater or equal to q.

Vector_2<Kernel> *operator-(p, q)*

returns the difference vector between q and p. You can substitute *ORIGIN* for either p or q, but not for both.

Point_2<Kernel> *operator+(p, Vector_2<Kernel> v)*

returns the point obtained by translating p by the vector v.

Point_2<Kernel> *operator-(p, Vector_2<Kernel> v)*

returns the point obtained by translating *p* by the vector -*v*.

Example

The following declaration creates two points with Cartesian double coordinates.

```
Point_2< Cartesian<double> > p, q(1.0, 2.0);
```

The variable `p` is uninitialized and should first be used on the left hand side of an assignment.

```
p = q;

std::cout << p.x() << "  " << p.y() << std::endl;
```

See Also

Kernel::Point_2 . . . . page 794

CGAL::Ray_2<Kernel>

Definition

An object r of the data type *Ray_2<Kernel>* is a directed straight ray in the two-dimensional Euclidean plane $\mathbb{E}^2$. It starts in a point called the *source* of r and goes to infinity.

Creation

Ray_2<Kernel> r(Point_2<Kernel> p, Point_2<Kernel> q);

introduces a ray r with source p and passing through point q.

Ray_2<Kernel> r(Point_2<Kernel> p, Direction_2<Kernel> d);

introduces a ray r starting at source p with direction d.

Ray_2<Kernel> r(Point_2<Kernel> p, Vector_2<Kernel> v);

introduces a ray r starting at source p with the direction of v.

Ray_2<Kernel> r(Point_2<Kernel> p, Line_2<Kernel> l);

introduces a ray r starting at source p with the same direction as l.

Operations

bool *r.operator==(h) const*

Test for equality: two rays are equal, iff they have the same source and the same direction.

bool *r.operator!=(h) const*

Test for inequality.

Point_2<Kernel>	*r.source() const*	returns the source of r.
Point_2<Kernel>	*r.point(int i) const*	returns a point on r. *point(0)* is the source, *point(i)*, with $i > 0$, is different from the source. *Precondition*: $i \geq 0$.

Direction_2<Kernel>	*r.direction() const*	returns the direction of *r*.
Vector_2<Kernel>	*r.to_vector() const*	returns a vector giving the direction of *r*.
Line_2<Kernel>	*r.supporting_line() const*	
		returns the line supporting *r* which has the same direction.
Ray_2<Kernel>	*r.opposite() const*	returns the ray with the same source and the opposite direction.

Predicates

bool	*r.is_degenerate() const*	
		ray *r* is degenerate, if the source and the second defining point fall together (that is if the direction is degenerate).
bool	*r.is_horizontal() const*	
bool	*r.is_vertical() const*	
bool	*r.has_on(Point_2<Kernel> p) const*	
		A point is on *r*, iff it is equal to the source of *r*, or if it is in the interior of *r*.
bool	*r.collinear_has_on(Point_2<Kernel> p) const*	
		checks if point *p* is on *r*. This function is faster than function *has_on()* if the precondition checking is disabled. *Precondition*: *p* is on the supporting line of *r*.

Miscellaneous

Ray_2<Kernel>	*r.transform(Aff_transformation_2<Kernel> t) const*	
		returns the ray obtained by applying *t* on the source and on the direction of *r*.

See Also

Kernel::Ray_2 . . . . page 798

CGAL::Segment_2<Kernel>

Class

Definition

An object s of the data type *Segment_2<Kernel>* is a directed straight line segment in the two-dimensional Euclidean plane $\mathbb{E}^2$, i.e. a straight line segment $[p,q]$ connecting two points $p,q \in \mathbb{R}^2$. The segment is topologically closed, i.e. the end points belong to it. Point p is called the *source* and q is called the *target* of s. The length of s is the Euclidean distance between p and q. Note that there is only a function to compute the square of the length, because otherwise we had to perform a square root operation which is not defined for all number types, which is expensive, and may not be exact.

Creation

Segment_2<Kernel> s(Point_2<Kernel> p, Point_2<Kernel> q);

introduces a segment s with source p and target q. The segment is directed from the source towards the target.

Operations

bool	*s.operator==(q) const*	Test for equality: Two segments are equal, iff their sources and targets are equal.
bool	*s.operator!=(q) const*	Test for inequality.
Point_2<Kernel>	*s.source() const*	returns the source of s.
Point_2<Kernel>	*s.target() const*	returns the target of s.
Point_2<Kernel>	*s.min() const*	returns the point of s with lexicographically smallest coordinate.
Point_2<Kernel>	*s.max() const*	returns the point of s with lexicographically largest coordinate.
Point_2<Kernel>	*s.vertex(int i) const*	returns source or target of s: *vertex(0)* returns the source of s, *vertex(1)* returns the target of s. The parameter i is taken modulo 2, which gives easy access to the other vertex.
Point_2<Kernel>	*s.point(int i) const*	returns *vertex(i)*.
Point_2<Kernel>	*s.operator[ ](int i) const*	returns *vertex(i)*.
Kernel::FT	*s.squared_length() const*	returns the squared length of s.
Direction_2<Kernel>	*s.direction() const*	returns the direction from source to target of s.

Vector_2<Kernel>	*s.to_vector() const*	returns the vector *s.target() - s.source()*.
Segment_2<Kernel>	*s.opposite() const*	returns a segment with source and target point interchanged.
Line_2<Kernel>	*s.supporting_line() const*	returns the line l passing through s. Line l has the same orientation as segment s.

Predicates

bool	*s.is_degenerate() const*	segment s is degenerate, if source and target are equal.
bool	*s.is_horizontal() const*	
bool	*s.is_vertical() const*	
bool	*s.has_on(Point_2<Kernel> p) const*	A point is on s, iff it is equal to the source or target of s, or if it is in the interior of s.
bool	*s.collinear_has_on(Point_2<Kernel> p) const*	checks if point p is on segment s. This function is faster than function *has_on()*. *Precondition*: p is on the supporting line of s.

Miscellaneous

Bbox_2	*s.bbox() const*	returns a bounding box containing s.
Segment_2<Kernel>	*s.transform(Aff_transformation_2<Kernel> t) const*	returns the segment obtained by applying t on the source and the target of s.

See Also

Kernel::Segment_2 . . . . page 800

CGAL::Triangle_2<Kernel>

Class

Definition

An object t of the class *Triangle_2<Kernel>* is a triangle in the two-dimensional Euclidean plane $\mathbb{E}^2$. Triangle t is oriented, i.e., its boundary has clockwise or counterclockwise orientation. We call the side to the left of the boundary the positive side and the side to the right of the boundary the negative side.

The boundary of a triangle splits the plane in two open regions, a bounded one and an unbounded one.

Creation

Triangle_2<Kernel> t(Point_2<Kernel> p, Point_2<Kernel> q, Point_2<Kernel> r);

introduces a triangle t with vertices p, q and r.

Operations

bool	*t.operator==(t2) const*	Test for equality: two triangles are equal, iff there exists a cyclic permutation of the vertices of $t2$, such that they are equal to the vertices of t.
bool	*t.operator!=(t2) const*	Test for inequality.
Point_2<Kernel>	*t.vertex(int i) const*	returns the i'th vertex modulo 3 of t.
Point_2<Kernel>	*t.operator[ ](int i) const*	returns *vertex(i)*.

Predicates

bool	*t.is_degenerate() const*	triangle t is degenerate, if the vertices are collinear.
Orientation	*t.orientation() const*	returns the orientation of t.

Oriented_side	*t.oriented_side(Point_2<Kernel> p) const*	
		returns *ON_ORIENTED_BOUNDARY*, or *POSITIVE_SIDE*, or the constant *ON_NEGATIVE_SIDE*, determined by the position of point *p*. *Precondition*: *t* is not degenerate.
Bounded_side	*t.bounded_side(Point_2<Kernel> p) const*	
		returns the constant *ON_BOUNDARY*, *ON_BOUNDED_SIDE*, or else *ON_UNBOUNDED_SIDE*, depending on where point *p* is. *Precondition*: *t* is not degenerate.

For convenience we provide the following Boolean functions:

bool	*t.has_on_positive_side(Point_2<Kernel> p) const*	
bool	*t.has_on_negative_side(Point_2<Kernel> p) const*	
bool	*t.has_on_boundary(Point_2<Kernel> p) const*	
bool	*t.has_on_bounded_side(Point_2<Kernel> p) const*	
bool	*t.has_on_unbounded_side(Point_2<Kernel> p) const*	
		Precondition: *t* is not degenerate.

Miscellaneous

Triangle_2<Kernel>	*t.opposite()*	returns a triangle where the boundary is oriented the other way round (this flips the positive and the negative side, but not the bounded and unbounded side).
Kernel::FT	*t.area() const*	returns the signed area of *t*.
Bbox_2	*t.bbox() const*	returns a bounding box containing *t*.
Triangle_2<Kernel>	*t.transform(Aff_transformation_2<Kernel> at) const*	
		returns the triangle obtained by applying *at* on the three vertices of *t*.

See Also

Kernel::Triangle_2 . . . . page 804

CGAL::Vector_2<Kernel>

Class

Definition

An object of the class *Vector_2<Kernel>* is a vector in the two-dimensional vector space $\mathbb{R}^2$. Geometrically spoken, a vector is the difference of two points p_2, p_1 and denotes the direction and the distance from p_1 to p_2.

CGAL defines a symbolic constant *NULL_VECTOR*. We will explicitly state where you can pass this constant as an argument instead of a vector initialized with zeros.

Types

Vector_2<Kernel>:: Cartesian_const_iterator

An iterator for enumerating the Cartesian coordinates of a vector.

Creation

Vector_2<Kernel> v(Point_2<Kernel> a, Point_2<Kernel> b);

introduces the vector $b-a$.

Vector_2<Kernel> v(Segment_2<Kernel> s);

introduces the vector $s.target() - s.source()$.

Vector_2<Kernel> v(Ray_2<Kernel> r);

introduces the vector having the same direction as r.

Vector_2<Kernel> v(Line_2<Kernel> l);

introduces the vector having the same direction as l.

Vector_2<Kernel> v(Null_vector NULL_VECTOR);

introduces a null vector v.

Vector_2<Kernel> v(int x, int y); introduces a vector v initialized to (x,y).

Vector_2<Kernel> v(double x, double y);

introduces a vector v initialized to (x,y).

Vector_2<Kernel> v(Kernel::RT hx, Kernel::RT hy, Kernel::RT hw = RT(1));

introduces a vector v initialized to $(hx/hw, hy/hw)$.
Precondition: $hw \neq 0$

Vector_2<Kernel> v(Kernel::FT x, Kernel::FT y);

introduces a vector v initialized to (x, y).

Operations

bool *v.operator==(w) const*

Test for equality: two vectors are equal, iff their x and y coordinates are equal. You can compare a vector with the *NULL_VECTOR*.

bool *v.operator!=(w) const*

Test for inequality. You can compare a vector with the *NULL_VECTOR*.

There are two sets of coordinate access functions, namely to the homogeneous and to the Cartesian coordinates. They can be used independently from the chosen kernel model.

Kernel::RT *v.hx() const*

returns the homogeneous x coordinate.

Kernel::RT *v.hy() const*

returns the homogeneous y coordinate.

Kernel::RT *v.hw() const*

returns the homogenizing coordinate.

Note that you do not loose information with the homogeneous representation, because the *FieldNumberType* is a quotient.

Kernel::FT *v.x() const*

returns the x-coordinate of v, that is hx/hw.

Kernel::FT *v.y() const*

returns the y-coordinate of v, that is hy/hw.

The following operations are for convenience and for compatibility with higher dimensional vectors. Again they come in a Cartesian and homogeneous flavor.

Kernel::RT *v.homogeneous(int i) const*

returns the i'th homogeneous coordinate of v, starting with 0.
Precondition: $0 \leq i \leq 2$.

Kernel::FT *v.cartesian(int i) const*

returns the i'th Cartesian coordinate of v, starting at 0.
Precondition: $0 \leq i \leq 1$.

Kernel::FT *v.operator[](int i) const*

returns *cartesian(i)*.
Precondition: $0 \leq i \leq 1$.

Cartesian_const_iterator *v.cartesian_begin() const*

returns an iterator to the Cartesian coordinates of v, starting with the 0th coordinate.

Cartesian_const_iterator *v.cartesian_end() const*

returns an off the end iterator to the Cartesian coordinates of v.

int *v.dimension() const*

returns the dimension (the constant 2).

Direction_2<Kernel> *v.direction() const*

returns the direction which passes through v.

Vector_2<Kernel> *v.transform(Aff_transformation_2<Kernel> t) const*

returns the vector obtained by applying t on v.

Vector_2<Kernel> *v.perpendicular(Orientation o) const*

returns the vector perpendicular to v in clockwise or counterclockwise orientation.

Operators

The following operations can be applied to vectors:

Vector_2<Kernel> *v.operator+(w) const*

Addition.

Vector_2<Kernel> *v.operator-(w) const*

Subtraction.

Vector_2<Kernel> *v.operator-() const*

returns the opposite vector.

Kernel::FT *v.operator*(w) const*

returns the scalar product (= inner product) of the two vectors.

Vector_2<Kernel> *operator*(v, Kernel::RT s)*

Multiplication with a scalar from the right.

Vector_2<Kernel> *operator*(v, Kernel::FT s)*

Multiplication with a scalar from the right.

Vector_2<Kernel> *operator*(Kernel::RT s, v)*

Multiplication with a scalar from the left.

Vector_2<Kernel> *operator*(Kernel::FT s, v)*

Multiplication with a scalar from the left.

Vector_2<Kernel> *v.operator/(Kernel::RT s) const*

Division by a scalar.

Kernel::FT *v.squared_length() const*

returns the squared length of *v*.

See Also

Kernel::Vector_2 . . . . page 806

11.11.2 Three-dimensional Objects

CGAL::Bbox_3

#include <CGAL/Bbox_3.h>

Definition

An object *b* of the class *Bbox_3* is a bounding box in the three-dimensional Euclidean space $\mathbb{E}^3$.

Creation

Bbox_3 b(double x_min, double y_min, double z_min, double x_max, double y_max, double z_max);

introduces a bounding box *b* with lexicographically smallest corner point at *(xmin, ymin, zmin)* lexicographically largest corner point at *(xmax, ymax, zmax)*.

Operations

bool	*b.operator==(c) const*	Test for equality.
bool	*b.operator!=(q) const*	Test for inequality.
int	*b.dimension() const*	Returns 3.
double	*b.xmin() const*	
double	*b.ymin() const*	
double	*b.zmin() const*	
double	*b.xmax() const*	
double	*b.ymax() const*	
double	*b.zmax() const*	
double	*b.min(int i) const*	Returns *xmin()* if *i==0* or *ymin()* if *i==1* or *zmin()* if *i==2*. *Precondition*: i¡=0 and i¡=2
double	*b.max(int i) const*	Returns *xmax()* if *i==0* or *ymax()* if *i==1* or *zmax()* if *i==2*. *Precondition*: i==0 and i¡=2
Bbox_3	*b.operator+(c) const*	returns a bounding box of *b* and *c*.

See Also

CGAL::Bbox_2 . . . . page 613
CGAL::do_overlap . . . . page 732

CGAL::Aff_transformation_3<Kernel>

Definition

The class *Aff_transformation_3<Kernel>* represents three-dimensional affine transformations. The general form of an affine transformation is based on a homogeneous representation of points. Thereby all transformations can be realized by matrix multiplication.

Multiplying the transformation matrix by a scalar does not change the represented transformation. Therefore, any transformation represented by a matrix with rational entries can be represented by a transformation matrix with integer entries as well. (Multiply the matrix with the common denominator of the rational entries.) Hence, it is sufficient to use the number type *Kernel::RT* to represent the entries of the transformation matrix.

CGAL offers several specialized affine transformations. Different constructors are provided to create them. They are parameterized with a symbolic name to denote the transformation type, followed by additional parameters. The symbolic name tags solve ambiguities in the function overloading and they make the code more readable, i.e., what type of transformation is created.

In three-dimensional space we have a 4×4 matrix $(m_{ij})_{i,\,j=0\ldots3}$. Entries m_{30}, m_{31}, and m_{32} are always zero and therefore do not appear in the constructors.

Creation

Aff_transformation_3<Kernel> t(Identity_transformation);

introduces an identity transformation.

Aff_transformation_3<Kernel> t(const Translation, Vector_3<Kernel> v);

introduces a translation by a vector v.

Aff_transformation_3<Kernel> t(const Scaling, Kernel::RT s, Kernel::RT hw = RT(1));

introduces a scaling by a scale factor s/hw.

Aff_transformation_3<Kernel> t(Kernel::RT m00,
Kernel::RT m01,
Kernel::RT m02,
Kernel::RT m03,
Kernel::RT m10,
Kernel::RT m11,
Kernel::RT m12,
Kernel::RT m13,
Kernel::RT m20,
Kernel::RT m21,
Kernel::RT m22,
Kernel::RT m23,

Kernel::RT hw = RT(1))

introduces a general affine transformation of the matrix form $\begin{pmatrix} m_{00} & m_{01} & m_{02} & m_{03} \\ m_{10} & m_{11} & m_{12} & m_{13} \\ m_{20} & m_{21} & m_{22} & m_{23} \\ 0 & 0 & 0 & hw \end{pmatrix}$. The part $\frac{1}{hw}$ $\begin{pmatrix} m_{00} & m_{01} & m_{02} \\ m_{10} & m_{11} & m_{12} \\ m_{20} & m_{21} & m_{22} \end{pmatrix}$ defines the scaling and rotational part of the transformation, while the vector $\frac{1}{hw}\begin{pmatrix} m_{03} \\ m_{13} \\ m_{23} \end{pmatrix}$ contains the translational part.

Aff_transformation_3<Kernel> t(Kernel::RT m00,
Kernel::RT m01,
Kernel::RT m02,
Kernel::RT m10,
Kernel::RT m11,
Kernel::RT m12,
Kernel::RT m20,
Kernel::RT m21,
Kernel::RT m22,
Kernel::RT hw = RT(1))

introduces a general linear transformation of the matrix form $\begin{pmatrix} m_{00} & m_{01} & m_{02} & 0 \\ m_{10} & m_{11} & m_{12} & 0 \\ m_{20} & m_{21} & m_{22} & 0 \\ 0 & 0 & 0 & hw \end{pmatrix}$, i.e. an affine transformation without translational part.

Operations

Each class *Class_3<Kernel>* representing a geometric object in 3D has a member function:

Class_3<Kernel> transform(Aff_transformation_3<Kernel> t).

The transformation classes provide a member function *transform()* for points, vectors, directions, and planes:

Point_3<Kernel>	*t.transform(Point_3<Kernel> p) const*
Vector_3<Kernel>	*t.transform(Vector_3<Kernel> p) const*
Direction_3<Kernel>	*t.transform(Direction_3<Kernel> p) const*
Plane_3<Kernel>	*t.transform(Plane_3<Kernel> p) const*

CGAL provides four function operators for these member functions:

Point_3<Kernel>	*t.operator()(Point_3<Kernel> p) const*
Vector_3<Kernel>	*t.operator()(Vector_3<Kernel> p) const*
Direction_3<Kernel>	*t.operator()(Direction_3<Kernel> p) const*
Plane_3<Kernel>	*t.operator()(Plane_3<Kernel> p) const*

Aff_transformation_3<Kernel>		
	t.operator(s) const*	composes two affine transformations.
Aff_transformation_3<Kernel>		
	t.inverse() const	gives the inverse transformation.
bool	*t.is_even() const*	returns *true*, if the transformation is not reflecting, i.e. the determinant of the involved linear transformation is non-negative.
bool	*t.is_odd() const*	returns *true*, if the transformation is reflecting.

The matrix entries of a matrix representation of a *Aff_transformation_3<Kernel>* can be accessed trough the following member functions:

Kernel::FT	*t.cartesian(int i, int j) const*	
Kernel::FT	*t.m(int i, int j) const*	returns entry m_{ij} in a matrix representation in which m_{33} is 1.
Kernel::RT	*t.homogeneous(int i, int j) const*	
Kernel::RT	*t.hm(int i, int j) const*	returns entry m_{ij} in some fixed matrix representation.

For affine transformations no I/O operators are defined.

See Also

CGAL::Aff_transformation_2<Kernel> . . . . page 609
CGAL::Identity_transformation . . . . page 686
CGAL::Reflection . . . . page 686
CGAL::Rotation . . . . page 687
CGAL::Scaling . . . . page 687
CGAL::Translation . . . . page 688

CGAL::Direction_3<Kernel>

Definition

An object of the class *Direction_3<Kernel>* is a vector in the three-dimensional vector space $\mathbb{R}^3$ where we forget about their length. They can be viewed as unit vectors, although there is no normalization internally, since this is error prone. Directions are used whenever the length of a vector does not matter. They also characterize a set of parallel lines that have the same orientation or the direction normal to parallel planes that have the same orientation. For example, you can ask for the direction orthogonal to an oriented plane, or the direction of an oriented line.

Creation

Direction_3<Kernel> d(Vector_3<Kernel> v);

introduces a direction d initialized with the direction of vector v.

Direction_3<Kernel> d(Line_3<Kernel> l);

introduces the direction d of line l.

Direction_3<Kernel> d(Ray_3<Kernel> r);

introduces the direction d of ray r.

Direction_3<Kernel> d(Segment_3<Kernel> s);

introduces the direction d of segment s.

Direction_3<Kernel> d(Kernel::RT x, Kernel::RT y, Kernel::RT z);

introduces a direction d initialized with the direction from the origin to the point with Cartesian coordinates (x, y, z).

Operations

Kernel::RT	*d.delta(int i) const*	returns values, such that *d== Direction_3<Kernel> (delta(0),delta(1),delta(2))*. *Precondition*: : $0 \leq i \leq 2$.
Kernel::RT	*d.dx() const*	returns *delta(0)*.
Kernel::RT	*d.dy() const*	returns *delta(1)*.
Kernel::RT	*d.dz() const*	returns *delta(2)*.
bool	*d.operator==(e) const*	Test for equality.
bool	*d.operator!=(e) const*	Test for inequality.
Direction_3<Kernel>	*d.operator-() const*	The direction opposite to d.
Vector_3<Kernel>	*d.vector() const*	returns a vector that has the same direction as d.
Direction_3<Kernel>	*d.transform(Aff_transformation_3<Kernel> t) const*	returns the direction obtained by applying t on d.

See Also

Kernel::Direction_3 . . . . page 786

Class

CGAL::Iso_cuboid_3<Kernel>

Definition

An object s of the data type *Iso_cuboid_3<Kernel>* is a cuboid in the Euclidean space $\mathbb{E}^3$ with edges parallel to the x, y and z axis of the coordinate system.

Although they are represented in a canonical form by only two vertices, namely the lexicographically smallest and largest vertex with respect to Cartesian xyz coordinates, we provide functions for "accessing" the other vertices as well.

Iso-oriented cuboids and bounding boxes are quite similar. The difference however is that bounding boxes have always double coordinates, whereas the coordinate type of an iso-oriented cuboid is chosen by the user.

Creation

Iso_cuboid_3<Kernel> c(Point_3<Kernel> p, Point_3<Kernel> q);

introduces an iso-oriented cuboid c with diagonal opposite vertices p and q. Note that the object is brought in the canonical form.

Iso_cuboid_3<Kernel> c(Point_3<Kernel> p, Point_3<Kernel> q, int);

introduces an iso-oriented cuboid c with diagonal opposite vertices p and q. The *int* argument value is only used to distinguish the two overloaded functions.
Precondition: $p.x() <= q.x()$, $p.y() <= q.y()$ and $p.z() <= q.z()$.

Iso_cuboid_3<Kernel> c(Point_3<Kernel> left,
Point_3<Kernel> right,
Point_3<Kernel> bottom,
Point_3<Kernel> top,
Point_3<Kernel> far,
Point_3<Kernel> close)

introduces an iso-oriented cuboid c whose minimal x coordinate is the one of *left*, the maximal x coordinate is the one of *right*, the minimal y coordinate is the one of *bottom*, the maximal y coordinate is the one of *top*, the minimal z coordinate is the one of *far*, the maximal z coordinate is the one of *close*.

Iso_cuboid_3<Kernel> c(Kernel::RT min_hx,
Kernel::RT min_hy,
Kernel::RT min_hz,
Kernel::RT max_hx,
Kernel::RT max_hy,
Kernel::RT max_hz,

Kernel::RT hw = RT(1))

introduces an iso-oriented cuboid *c* with diagonal opposite vertices (*min_hx/hw*, *min_hy/hw*, *min_hz/hw*) and (*max_hx/hw*, *max_hy/hw*, *max_hz/hw*).
Precondition: $hw \neq 0$.

Operations

bool *c.operator==(c2) const*

Test for equality: two iso-oriented cuboid are equal, iff their lower left and their upper right vertices are equal.

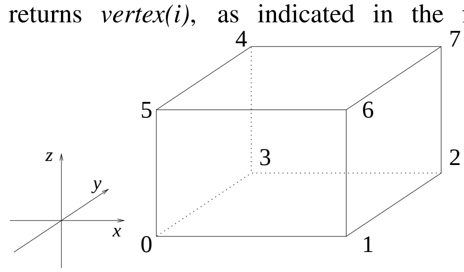
bool *c.operator!=(c2) const*

Test for inequality.

Point_3<Kernel> *c.vertex(int i) const* returns the i'th vertex modulo 8 of *c*. starting with the lower left vertex.

Point_3<Kernel> *c.operator[ ](int i) const*

returns *vertex(i)*, as indicated in the figure below:



Point_3<Kernel> *c.min() const* returns the smallest vertex of *c* (= *vertex(0)*).

Point_3<Kernel> *c.max() const* returns the largest vertex of *c* (= *vertex(7)*).

Kernel::FT	*c.xmin() const*	returns smallest Cartesian *x*-coordinate in *c*.
Kernel::FT	*c.ymin() const*	returns smallest Cartesian *y*-coordinate in *c*.
Kernel::FT	*c.zmin() const*	returns smallest Cartesian *z*-coordinate in *c*.
Kernel::FT	*c.xmax() const*	returns largest Cartesian *x*-coordinate in *c*.
Kernel::FT	*c.ymax() const*	returns largest Cartesian *y*-coordinate in *c*.
Kernel::FT	*c.zmax() const*	returns largest Cartesian *z*-coordinate in *c*.

Kernel::FT *c.min_coord(int i) const*

returns *i*-th Cartesian coordinate of the smallest vertex of *c*.
Precondition: $0 \leq i \leq 2$.

Kernel::FT	*c.max_coord(int i) const*	
		returns i-th Cartesian coordinate of the largest vertex of c. *Precondition*: $0 \leq i \leq 2$.

Predicates

bool	*c.is_degenerate() const*	
		c is degenerate, if all vertices are coplanar.
Bounded_side	*c.bounded_side(Point_3<Kernel> p) const*	
		returns either *ON_UNBOUNDED_SIDE*, *ON_BOUNDED_SIDE*, or the constant *ON_BOUNDARY*, depending on where point p is.
bool	*c.has_on_boundary(Point_3<Kernel> p) const*	
bool	*c.has_on_bounded_side(Point_3<Kernel> p) const*	
bool	*c.has_on_unbounded_side(Point_3<Kernel> p) const*	

Miscellaneous

Kernel::FT	*c.volume() const*	returns the volume of c.
Bbox_3	*c.bbox() const*	returns a bounding box containing c.
Iso_cuboid_3<Kernel>	*c.transform(Aff_transformation_3<Kernel> t) const*	
		returns the iso-oriented cuboid obtained by applying t on the smallest and the largest of c. *Precondition*: The angle at a rotation must be a multiple of $\pi/2$, otherwise the resulting cuboid does not have the same size. Note that rotating about an arbitrary angle can even result in a degenerate iso-oriented cuboid.

See Also

Kernel::IsoCuboid_3 . . . . page 787

CGAL::Line_3<Kernel>

Class

Definition

An object l of the data type *Line_3<Kernel>* is a directed straight line in the three-dimensional Euclidean space $\mathbb{E}^3$.

Creation

Line_3<Kernel> l(Point_3<Kernel> p, Point_3<Kernel> q);

introduces a line l passing through the points p and q. Line l is directed from p to q.

Line_3<Kernel> l(Point_3<Kernel> p, Direction_3<Kernel> d);

introduces a line l passing through point p with direction d.

Line_3<Kernel> l(Point_3<Kernel> p, Vector_3<Kernel> v);

introduces a line l passing through point p and oriented by v.

Line_3<Kernel> l(Segment_3<Kernel> s); returns the line supporting the segment s, oriented from source to target.

Line_3<Kernel> l(Ray_3<Kernel> r); returns the line supporting the ray r, with the same orientation.

Operations

bool *l.operator==(h) const*

Test for equality: two lines are equal, iff they have a non empty intersection and the same direction.

bool *l.operator!=(h) const* Test for inequality.

Point_3<Kernel> *l.projection(Point_3<Kernel> p) const*

returns the orthogonal projection of p on l.

Point_3<Kernel> *l.point(int i) const* returns an arbitrary point on l. It holds *point(i) = point(j)*, iff *i=j*.

Predicates

bool	*l.is_degenerate() const*	
		returns *true* iff line *l* is degenerated to a point.
bool	*l.has_on(Point_3<Kernel> p) const*	
		returns *true* iff *p* lies on *l*.

Miscellaneous

Plane_3<Kernel>	*l.perpendicular_plane(Point_3<Kernel> p) const*	
		returns the plane perpendicular to *l* passing through *p*.
Line_3<Kernel>	*l.opposite() const*	returns the line with opposite direction.
Vector_3<Kernel>	*l.to_vector() const*	returns a vector having the same direction as *l*.
Direction_3<Kernel>	*l.direction() const*	returns the direction of *l*.
Line_3<Kernel>	*l.transform(Aff_transformation_3<Kernel> t) const*	
		returns the line obtained by applying *t* on a point on *l* and the direction of *l*.

See Also

Kernel::Line_3 . . . . page 790

CGAL::Plane_3<Kernel>

Class

Definition

An object h of the data type *Plane_3<Kernel>* is an oriented plane in the three-dimensional Euclidean space $\mathbb{E}^3$. It is defined by the set of points with Cartesian coordinates (x,y,z) that satisfy the plane equation

$$h: \; a\,x + b\,y + c\,z + d = 0.$$

The plane splits $\mathbb{E}^3$ in a *positive* and a *negative side*. A point p with Cartesian coordinates (px, py, pz) is on the positive side of h, iff $a\,px + b\,py + c\,pz + d > 0$. It is on the negative side, iff $a\,px + b\,py + c\,pz + d < 0$.

Creation

Plane_3<Kernel> h(Kernel::RT a, Kernel::RT b, Kernel::RT c, Kernel::RT d);

creates a plane h defined by the equation $a\,px + b\,py + c\,pz + d = 0$. Notice that h is degenerate if $a = b = c = 0$.

Plane_3<Kernel> h(Point_3<Kernel> p, Point_3<Kernel> q, Point_3<Kernel> r);

creates a plane h passing through the points p, q and r. The plane is oriented such that p, q and r are oriented in a positive sense (that is counterclockwise) when seen from the positive side of h. Notice that h is degenerate if the points are collinear.

Plane_3<Kernel> h(Point_3<Kernel> p, Vector_3<Kernel> v);

introduces a plane h that passes through point p and that is orthogonal to v.

Plane_3<Kernel> h(Point_3<Kernel> p, Direction_3<Kernel> d);

introduces a plane h that passes through point p and that has as an orthogonal direction equal to d.

Plane_3<Kernel> h(Line_3<Kernel> l, Point_3<Kernel> p);

introduces a plane h that is defined through the three points *l.point(0)*, *l.point(1)* and p.

Plane_3<Kernel> h(Ray_3<Kernel> r, Point_3<Kernel> p);

introduces a plane h that is defined through the three points *r.point(0)*, *r.point(1)* and p.

Plane_3<Kernel> h(Segment_3<Kernel> s, Point_3<Kernel> p);

introduces a plane *h* that is defined through the three points *s.source()*, *s.target()* and *p*.

Plane_3<Kernel> h(Circle_3<Kernel> c); introduces a plane *h* that is defined as the plane containing the circle.

Operations

bool *h.operator==(h2) const*

Test for equality: two planes are equal, iff they have a non empty intersection and the same orientation.

bool *h.operator!=(h2) const*

Test for inequality.

Kernel::RT	*h.a() const*	returns the first coefficient of *h*.
Kernel::RT	*h.b() const*	returns the second coefficient of *h*.
Kernel::RT	*h.c() const*	returns the third coefficient of *h*.
Kernel::RT	*h.d() const*	returns the fourth coefficient of *h*.

Line_3<Kernel> *h.perpendicular_line(Point_3<Kernel> p) const*

returns the line that is perpendicular to *h* and that passes through point *p*. The line is oriented from the negative to the positive side of *h*.

Point_3<Kernel> *h.projection(Point_3<Kernel> p) const*

returns the orthogonal projection of *p* on *h*.

Plane_3<Kernel> *h.opposite() const* returns the plane with opposite orientation.

Point_3<Kernel> *h.point() const* returns an arbitrary point on *h*.

Vector_3<Kernel> *h.orthogonal_vector() const*

returns a vector that is orthogonal to *h* and that is directed to the positive side of *h*.

Direction_3<Kernel> *h.orthogonal_direction() const*

returns the direction that is orthogonal to *h* and that is directed to the positive side of *h*.

Vector_3<Kernel>	*h.base1() const*	returns a vector orthogonal to *orthogonal_vector()*.
Vector_3<Kernel>	*h.base2() const*	returns a vector that is both orthogonal to *base1()*, and to *orthogonal_vector()*, and such that the result of *orientation(point(), point() + base1(), point()+base2(), point() + orthogonal_vector())* is positive.

2D Conversion

The following functions provide conversion between a plane and CGAL's two-dimensional space. The transformation is affine, but not necessarily an isometry. This means, the transformation preserves combinatorics, but not distances.

Point_2<Kernel> *h.to_2d(Point_3<Kernel> p) const*

returns the image point of the projection of p under an affine transformation, which maps h onto the xy-plane, with the z-coordinate removed.

Point_3<Kernel> *h.to_3d(Point_2<Kernel> p) const*

returns a point q, such that *to_2d(to_3d(p))* is equal to p.

Predicates

Oriented_side *h.oriented_side(Point_3<Kernel> p) const*

returns either *ON_ORIENTED_BOUNDARY*, or the constant *ON_POSITIVE_SIDE*, or the constant *ON_NEGATIVE_SIDE*, determined by the position of p relative to the oriented plane h.

For convenience we provide the following Boolean functions:

bool	*h.has_on(Point_3<Kernel> p) const*
bool	*h.has_on_positive_side(Point_3<Kernel> p) const*
bool	*h.has_on_negative_side(Point_3<Kernel> p) const*
bool	*h.has_on(Line_3<Kernel> l) const*
bool	*h.has_on(Circle_3<Kernel> l) const*

bool *h.is_degenerate() const*

Plane h is degenerate, if the coefficients a, b, and c of the plane equation are zero.

Miscellaneous

Plane_3<Kernel> *h.transform(Aff_transformation_3<Kernel> t) const*

returns the plane obtained by applying t on a point of h and the orthogonal direction of h.

See Also

Kernel::Plane_3 . . . . page 793

CGAL::Point_3<Kernel>

Class

Definition

An object of the class *Point_3<Kernel>* is a point in the three-dimensional Euclidean space $\mathbb{E}^3$.

Remember that *Kernel::RT* and *Kernel::FT* denote a RingNumberType and a FieldNumberType, respectively. For the kernel model *Cartesian<T>*, the two types are the same. For the kernel model *Homogeneous<T>*, *Kernel::RT* is equal to *T*, and *Kernel::FT* is equal to *Quotient<T>*.

Types

Point_3<Kernel>:: Cartesian_const_iterator

An iterator for enumerating the Cartesian coordinates of a point.

Creation

Point_3<Kernel> p(Origin ORIGIN);

introduces a point with Cartesian coordinates$(0,0,0)$.

Point_3<Kernel> p(int x, int y, int z);

introduces a point *p* initialized to (x,y,z).

Point_3<Kernel> p(double x, double y, double z);

introduces a point *p* initialized to (x,y,z) provided *RT* supports it.

Point_3<Kernel> p(Kernel::RT hx, Kernel::RT hy, Kernel::RT hz, Kernel::RT hw = RT(1));

introduces a point *p* initialized to $(hx/hw, hy/hw, hz/hw)$.
Precondition: $hw \neq 0$.

Point_3<Kernel> p(Kernel::FT x, Kernel::FT y, Kernel::FT z);

introduces a point *p* initialized to (x,y,z).

Operations

bool *p.operator==(q) const*

Test for equality: Two points are equal, iff their *x*, *y* and *z* coordinates are equal.

bool *p.operator!=(q) const*

Test for inequality.

There are two sets of coordinate access functions, namely to the homogeneous and to the Cartesian coordinates. They can be used independently from the chosen kernel model.

Kernel::RT *p.hx() const*

returns the homogeneous x coordinate.

Kernel::RT *p.hy() const*

returns the homogeneous y coordinate.

Kernel::RT *p.hz() const*

returns the homogeneous z coordinate.

Kernel::RT *p.hw() const*

returns the homogenizing coordinate.

Note that you do not loose information with the homogeneous representation, because the FieldNumberType is a quotient.

Kernel::FT *p.x() const*

returns the Cartesian x coordinate, that is hx/hw.

Kernel::FT *p.y() const*

returns the Cartesian y coordinate, that is hy/hw.

Kernel::FT *p.z() const*

returns the Cartesian z coordinate, that is hz/hw.

The following operations are for convenience and for compatibility with code for higher dimensional points. Again they come in a Cartesian and in a homogeneous flavor.

Kernel::RT *p.homogeneous(int i) const*

returns the i'th homogeneous coordinate of p, starting with 0.
Precondition: $0 \leq i \leq 3$.

Kernel::FT *p.cartesian(int i) const*

returns the i'th Cartesian coordinate of p, starting with 0.
Precondition: $0 \leq i \leq 2$.

Kernel::FT *p.operator[](int i) const*

returns *cartesian(i)*.
Precondition: $0 \leq i \leq 2$.

Cartesian_const_iterator *p.cartesian_begin() const*

returns an iterator to the Cartesian coordinates of *p*, starting with the 0th coordinate.

Cartesian_const_iterator *p.cartesian_end() const*

returns an off the end iterator to the Cartesian coordinates of *p*.

int *p.dimension() const*

returns the dimension (the constant 3).

Bbox_3 *p.bbox() const*

returns a bounding box containing *p*.

Point_3<Kernel> *p.transform(Aff_transformation_3<Kernel> t) const*

returns the point obtained by applying *t* on *p*.

Operators

The following operations can be applied on points:

bool *operator<(p, q)*

returns true iff *p* is lexicographically smaller than *q* (the lexicographical order being defined on the Cartesian coordinates).

bool *operator>(p, q)*

returns true iff *p* is lexicographically greater than *q*.

bool *operator<=(p, q)*

returns true iff *p* is lexicographically smaller or equal to *q*.

bool *operator>=(p, q)*

returns true iff *p* is lexicographically greater or equal to *q*.

Vector_3<Kernel> *operator-(p, q)*

returns the difference vector between *q* and *p*. You can substitute *ORIGIN* for either *p* or *q*, but not for both.

Point_3<Kernel> *operator+(p, Vector_3<Kernel> v)*

returns the point obtained by translating p by the vector v.

Point_3<Kernel> *operator-(p, Vector_3<Kernel> v)*

returns the point obtained by translating p by the vector -v.

See Also

Kernel::Point_3 . . . . page 796

CGAL::Ray_3<Kernel>

Class

Definition

An object r of the data type *Ray_3<Kernel>* is a directed straight ray in the three-dimensional Euclidean space $\mathbb{E}^3$. It starts in a point called the *source* of r and it goes to infinity.

Creation

Ray_3<Kernel> r(Point_3<Kernel> p, Point_3<Kernel> q);

introduces a ray r with source p and passing through point q.

Ray_3<Kernel> r(Point_3<Kernel> p, Direction_3<Kernel> d);

introduces a ray r with source p and with direction d.

Ray_3<Kernel> r(Point_3<Kernel> p, Vector_3<Kernel> v);

introduces a ray r with source p and with a direction given by v.

Ray_3<Kernel> r(Point_3<Kernel> p, Line_3<Kernel> l);

introduces a ray r starting at source p with the same direction as l.

Operations

bool *r.operator==(h) const*

Test for equality: two rays are equal, iff they have the same source and the same direction.

bool *r.operator!=(h) const*

Test for inequality.

Point_3<Kernel> *r.source() const* returns the source of r

Point_3<Kernel> *r.point(int i) const* returns a point on r. *point(0)* is the source. *point(i)*, with $i > 0$, is different from the source.
Precondition: $i \geq 0$.

Direction_3<Kernel>	*r.direction() const*	returns the direction of *r*.
Vector_3<Kernel>	*r.to_vector() const*	returns a vector giving the direction of *r*.
Line_3<Kernel>	*r.supporting_line() const*	returns the line supporting *r* which has the same direction.
Ray_3<Kernel>	*r.opposite() const*	returns the ray with the same source and the opposite direction.
bool	*r.is_degenerate() const*	ray *r* is degenerate, if the source and the second defining point fall together (that is if the direction is degenerate).
bool	*r.has_on(Point_3<Kernel> p) const*	A point is on *r*, iff it is equal to the source of *r*, or if it is in the interior of *r*.
Ray_3<Kernel>	*r.transform(Aff_transformation_3<Kernel> t) const*	returns the ray obtained by applying *t* on the source and on the direction of *r*.

See Also

Kernel::Ray_3 . . . . page 799

CGAL::Segment_3<Kernel>

Definition

An object s of the data type *Segment_3<Kernel>* is a directed straight line segment in the three-dimensional Euclidean space $\mathbb{E}^3$, i.e. a straight line segment $[p,q]$ connecting two points $p,q \in \mathbb{R}^3$. The segment is topologically closed, i.e. the end points belong to it. Point p is called the *source* and q is called the *target* of s. The length of s is the Euclidean distance between p and q. Note that there is only a function to compute the square of the length, because otherwise we had to perform a square root operation which is not defined for all number types, which is expensive, and may not be exact.

Creation

Segment_3<Kernel> s(Point_3<Kernel> p, Point_3<Kernel> q);

introduces a segment s with source p and target q. It is directed from the source towards the target.

Operations

bool	*s.operator==(q) const*	Test for equality: Two segments are equal, iff their sources and targets are equal.
bool	*s.operator!=(q) const*	Test for inequality.
Point_3<Kernel>	*s.source() const*	returns the source of s.
Point_3<Kernel>	*s.target() const*	returns the target of s.
Point_3<Kernel>	*s.min() const*	returns the point of s with smallest coordinate (lexicographically).
Point_3<Kernel>	*s.max() const*	returns the point of s with largest coordinate (lexicographically).
Point_3<Kernel>	*s.vertex(int i) const*	returns source or target of s: *vertex(0)* returns the source, *vertex(1)* returns the target. The parameter i is taken modulo 2, which gives easy access to the other vertex.
Point_3<Kernel>	*s.point(int i) const*	returns *vertex(i)*.
Point_3<Kernel>	*s.operator[](int i) const*	returns *vertex(i)*.

Kernel::FT	*s.squared_length() const*	
		returns the squared length of *s*.
Vector_3<Kernel>	*s.to_vector() const*	returns the vector *s.target() - s.source()*.
Direction_3<Kernel>	*s.direction() const*	returns the direction from source to target.
Segment_3<Kernel>	*s.opposite() const*	returns a segment with source and target interchanged.
Line_3<Kernel>	*s.supporting_line() const*	
		returns the line *l* passing through *s*. Line *l* has the same orientation as segment *s*, that is from the source to the target of *s*.
bool	*s.is_degenerate() const*	
		segment *s* is degenerate, if source and target fall together.
bool	*s.has_on(Point_3<Kernel> p) const*	
		A point is on *s*, iff it is equal to the source or target of *s*, or if it is in the interior of *s*.
Bbox_3	*s.bbox() const*	returns a bounding box containing *s*.
Segment_3<Kernel>	*s.transform(Aff_transformation_3<Kernel> t) const*	
		returns the segment obtained by applying *t* on the source and the target of *s*.

See Also

Kernel::Segment_3 . . . . page 801

CGAL::Circle_3<Kernel>

Class

Definition

An object of type *Circle_3<Kernel>* is a circle in the three-dimensional Euclidean space $\mathbb{E}^3$. Note that the circle can be degenerate, i.e. the squared radius may be zero.

Creation

Circle_3<Kernel> c(Point_3<Kernel> center, Kernel::FT sq_r, Plane_3<Kernel> plane);

introduces a variable *c* of type *Circle_3<Kernel>*. It is initialized to the circle of center *center* and squared radius *sq_r* in plane *plane*.
Precondition: *center* lies in *plane* and $sq\_r \geq 0$.

Circle_3<Kernel> c(Point_3<Kernel> center, Kernel::FT sq_r, Vector_3<Kernel> n);

introduces a variable *c* of type *Circle_3<Kernel>*. It is initialized to the circle of center *center* and squared radius *sq_r* in a plane normal to the vector *n*.
Precondition: $sq\_r \geq 0$.

Circle_3<Kernel> c(Point_3<Kernel> p, Point_3<Kernel> q, Point_3<Kernel> r);

introduces a variable *c* of type *Circle_3<Kernel>*. It is initialized to the circle passing through the three points.
Precondition: The three points are not collinear.

Circle_3<Kernel> c(Sphere_3<Kernel> sphere1, Sphere_3<Kernel> sphere2);

introduces a variable *c* of type *Circle_3<Kernel>*. It is initialized to the circle along which the two spheres intersect.
Precondition: The two spheres intersect along a circle.

Circle_3<Kernel> c(Sphere_3<Kernel> sphere, Plane_3<Kernel> plane);

introduces a variable *c* of type *Circle_3<Kernel>*. It is initialized to the circle along which the sphere and the plane intersect.
Precondition: The sphere and the plane intersect along a circle.

Circle_3<Kernel> c(Plane_3<Kernel> plane, Sphere_3<Kernel> sphere);

introduces a variable *c* of type *Circle_3<Kernel>*. It is initialized to the circle along which the sphere and the plane intersect.
Precondition: The sphere and the plane intersect along a circle.

Access Functions

Point_3<Kernel>	*c.center() const*	returns the center of *c*.
Kernel::FT	*c.squared_radius() const*	returns the squared radius of *c*.
Plane_3<Kernel>	*c.supporting_plane() const*	returns the supporting plane of *c*.
Sphere_3<Kernel>	*c.diametral_sphere() const*	returns the diametral sphere of *c*.
Kernel::FT	*c.area_divided_by_pi() const*	returns the area of *c*, divided by π.
double	*c.approximate_area() const*	returns an approximation of the area of *c*.
Kernel::FT	*c.squared_length_divided_by_pi_square() const*	returns the squared length of *c*, divided by π^2.
double	*c.approximate_squared_length() const*	returns an approximation of the squared length (i.e. perimeter) of *c*.

Predicates

bool	*c.has_on(Point_3<Kernel> p) const*	

Operations

bool operator	*==(c1, c2)*	returns *true*, iff *c1* and *c2* are equal, i.e. if they have the same center, the same squared radius and the same supporting plane.
bool operator	*!=(c1, c2)*	

Bbox_3	*c.bbox() const*	returns a bounding box containing c.

See Also

Kernel::Circle_3 . . . . page 784

CGAL::Sphere_3<Kernel>

Definition

An object of type *Sphere_3<Kernel>* is a sphere in the three-dimensional Euclidean space $\mathbb{E}^3$. The sphere is oriented, i.e. its boundary has clockwise or counterclockwise orientation. The boundary splits $\mathbb{E}^3$ into a positive and a negative side, where the positive side is to the left of the boundary. The boundary also splits $\mathbb{E}^3$ into a bounded and an unbounded side. Note that the sphere can be degenerated, i.e. the squared radius may be zero.

Creation

Sphere_3<Kernel> c(Point_3<Kernel> center,
Kernel::FT squared_radius,
Orientation orientation = COUNTERCLOCKWISE)

introduces a variable *c* of type *Sphere_3<Kernel>*. It is initialized to the sphere with center *center*, squared radius *squared_radius* and orientation *orientation*.
Precondition: *orientation* $\neq$ *COPLANAR*, and furthermore, *squared_radius* ≥ 0.

Sphere_3<Kernel> c(Point_3<Kernel> p, Point_3<Kernel> q, Point_3<Kernel> r, Point_3<Kernel> s);

introduces a variable *c* of type *Sphere_3<Kernel>*. It is initialized to the unique sphere which passes through the points *p*, *q*, *r* and *s*. The orientation of the sphere is the orientation of the point quadruple *p*, *q*, *r*, *s*.
Precondition: *p*, *q*, *r*, and *s* are not coplanar.

Sphere_3<Kernel> c(Point_3<Kernel> p,
Point_3<Kernel> q,
Point_3<Kernel> r,
Orientation o = COUNTERCLOCKWISE)

introduces a variable *c* of type *Sphere_3<Kernel>*. It is initialized to the smallest sphere which passes through the points *p*, *q*, and *r*. The orientation of the sphere is *o*.
Precondition: *o* is not *COPLANAR*.

Sphere_3<Kernel> c(Point_3<Kernel> p, Point_3<Kernel> q, Orientation o = COUNTERCLOCKWISE);

introduces a variable *c* of type *Sphere_3<Kernel>*. It is initialized to the smallest sphere which passes through the points *p* and *q*. The orientation of the sphere is *o*.
Precondition: *o* is not *COPLANAR*.

Sphere_3<Kernel> c(Point_3<Kernel> center, Orientation orientation = COUNTERCLOCKWISE);

introduces a variable *c* of type *Sphere_3<Kernel>*. It is initialized to the sphere with center *center*, squared radius zero and orientation *orientation*.
Precondition: *orientation* $\neq$ *COPLANAR*.
Postcondition: *c.is_degenerate() = true*.

Sphere_3<Kernel> c(Circle_3<Kernel> c);

introduces a variable *c* of type *Sphere_3<Kernel>*. It is initialized to the diametral sphere of the circle.

Access Functions

Point_3<Kernel> *c.center() const*

returns the center of *c*.

Kernel::FT *c.squared_radius() const*

returns the squared radius of *c*.

Orientation *c.orientation() const*

returns the orientation of *c*.

bool operator *c.==(sphere2) const*

returns *true*, iff *c* and *sphere2* are equal, i.e. if they have the same center, same squared radius and same orientation.

bool operator *c.!=(sphere2) const*

returns *true*, iff *c* and *sphere2* are not equal.

Predicates

bool *c.is_degenerate() const*

returns *true*, iff *c* is degenerate, i.e. if *c* has squared radius zero.

Oriented_side *c.oriented_side(Point_3<Kernel> p) const*

returns either the constant *ON_ORIENTED_BOUNDARY*, *ON_POSITIVE_SIDE*, or *ON_NEGATIVE_SIDE*, iff *p* lies on the boundary, properly on the positive side, or properly on the negative side of *c*, resp.

Bounded_side *c.bounded_side(Point_3<Kernel> p) const*

returns *ON_BOUNDED_SIDE*, *ON_BOUNDARY*, or *ON_UNBOUNDED_SIDE* iff *p* lies properly inside, on the boundary, or properly outside of *c*, resp.

bool *c.has_on_positive_side(Point_3<Kernel> p) const*
bool *c.has_on_negative_side(Point_3<Kernel> p) const*
bool *c.has_on_boundary(Point_3<Kernel> p) const*

bool	*c.has_on_bounded_side(Point_3<Kernel> p) const*	
bool	*c.has_on_unbounded_side(Point_3<Kernel> p) const*	
bool	*c.has_on(Point_3<Kernel> p) const*	
bool	*c.has_on(Circle_3<Kernel> p) const*	

Miscellaneous

Sphere_3<Kernel>	*c.opposite() const*	returns the sphere with the same center and squared radius as c but with opposite orientation.
Sphere_3<Kernel>	*c.orthogonal_transform(Aff_transformation_3<Kernel> at) const*	returns the sphere obtained by applying *at* on c. *Precondition*: *at* is an orthogonal transformation.
Bbox_3	*c.bbox() const*	returns a bounding box containing c.

See Also

Kernel::Sphere_3 . . . . page 802

CGAL::Tetrahedron_3<Kernel>

Class

Definition

An object t of the class *Tetrahedron_3<Kernel>* is an oriented tetrahedron in the three-dimensional Euclidean space $\mathbb{E}^3$.

It is defined by four vertices p_0, p_1, p_2 and p_3. The orientation of a tetrahedron is the orientation of its four vertices. That means it is positive when p_3 is on the positive side of the plane defined by p_0, p_1 and p_2.

The tetrahedron itself splits the space $\mathbb{E}^3$ in a *positive* and a *negative* side.

The boundary of a tetrahedron splits the space in two open regions, a bounded one and an unbounded one.

Creation

Tetrahedron_3<Kernel> t(Point_3<Kernel> p0, Point_3<Kernel> p1, Point_3<Kernel> p2, Point_3<Kernel> p3)

introduces a tetrahedron t with vertices p_0, p_1, p_2 and p_3.

Operations

bool *t.operator==(t2) const*

Test for equality: two tetrahedra t and $t2$ are equal, iff t and $t2$ have the same orientation and their sets (not sequences) of vertices are equal.

bool *t.operator!=(t2) const*

Test for inequality.

Point_3<Kernel> *t.vertex(int i) const* returns the i'th vertex modulo 4 of t.

Point_3<Kernel> *t.operator[](int i) const*

returns *vertex(int i)*.

Predicates

bool *t.is_degenerate() const*

Tetrahedron t is degenerate, if the vertices are coplanar.

Orientation	*t.orientation() const*	
Oriented_side	*t.oriented_side(Point_3<Kernel> p) const*	
		Precondition: : *t* is not degenerate.
Bounded_side	*t.bounded_side(Point_3<Kernel> p) const*	
		Precondition: : *t* is not degenerate.

For convenience we provide the following Boolean functions:

bool	*t.has_on_positive_side(Point_3<Kernel> p) const*
bool	*t.has_on_negative_side(Point_3<Kernel> p) const*
bool	*t.has_on_boundary(Point_3<Kernel> p) const*
bool	*t.has_on_bounded_side(Point_3<Kernel> p) const*
bool	*t.has_on_unbounded_side(Point_3<Kernel> p) const*

Miscellaneous

Kernel::FT	*t.volume() const*	returns the signed volume of *t*.
Bbox_3	*t.bbox() const*	returns a bounding box containing *t*.
Tetrahedron_3<Kernel>	*t.transform(Aff_transformation_3<Kernel> at) const*	
		returns the tetrahedron obtained by applying *at* on the three vertices of *t*.

See Also

Kernel::Tetrahedron_3 . . . . page 803

CGAL::Triangle_3<Kernel>

Class

Definition

An object t of the class *Triangle_3<Kernel>* is a triangle in the three-dimensional Euclidean space $\mathbb{E}^3$. As the triangle is not a full-dimensional object there is only a test whether a point lies on the triangle or not.

Creation

Triangle_3<Kernel> t(Point_3<Kernel> p, Point_3<Kernel> q, Point_3<Kernel> r);

introduces a triangle t with vertices p, q and r.

Operations

bool	*t.operator==(t2) const*	
		Test for equality: two triangles t and t_2 are equal, iff there exists a cyclic permutation of the vertices of $t2$, such that they are equal to the vertices of t.
bool	*t.operator!=(t2) const*	
		Test for inequality.
Point_3<Kernel>	*t.vertex(int i) const*	returns the i'th vertex modulo 3 of t.
Point_3<Kernel>	*t.operator[](int i) const*	
		returns *vertex(int i)*.
Plane_3<Kernel>	*t.supporting_plane()*	returns the supporting plane of t, with same orientation.

Predicates

bool	*t.is_degenerate() const*	
		t is degenerate if its vertices are collinear.
bool	*t.has_on(Point_3<Kernel> p) const*	
		A point is on t, if it is on a vertex, an edge or the face of t.

Miscellaneous

Kernel::FT	*t.squared_area() const*	
		returns a square of the area of *t*.
Bbox_3	*t.bbox() const*	returns a bounding box containing *t*.
Triangle_3<Kernel>	*t.transform(Aff_transformation_3<Kernel> at) const*	
		returns the triangle obtained by applying *at* on the three vertices of *t*.

See Also

Kernel::Triangle_3 . . . . page 805

CGAL::Vector_3<Kernel>

Definition

An object of the class *Vector_3<Kernel>* is a vector in the three-dimensional vector space $\mathbb{R}^3$. Geometrically spoken a vector is the difference of two points p_2, p_1 and denotes the direction and the distance from p_1 to p_2.

CGAL defines a symbolic constant *NULL_VECTOR*. We will explicitly state where you can pass this constant as an argument instead of a vector initialized with zeros.

Types

Vector_3<Kernel>:: Cartesian_const_iterator

An iterator for enumerating the Cartesian coordinates of a vector.

Creation

Vector_3<Kernel> v(Point_3<Kernel> a, Point_3<Kernel> b);

introduces the vector $b-a$.

Vector_3<Kernel> v(Segment_3<Kernel> s);

introduces the vector $s.target() - s.source()$.

Vector_3<Kernel> v(Ray_3<Kernel> r);

introduces a vector having the same direction as r.

Vector_3<Kernel> v(Line_3<Kernel> l);

introduces a vector having the same direction as l.

Vector_3<Kernel> v(Null_vector NULL_VECTOR);

introduces a null vector v.

Vector_3<Kernel> v(int x, int y, int z);

introduces a vector v initialized to (x,y,z).

Vector_3<Kernel> v(double x, double y, double z);

introduces a vector v initialized to (x,y,z).

Vector_3<Kernel> v(Kernel::RT hx, Kernel::RT hy, Kernel::RT hz, Kernel::RT hw = RT(1));

introduces a vector v initialized to $(hx/hw, hy/hw, hz/hw)$.

Vector_3<Kernel> v(Kernel::FT x, Kernel::FT y, Kernel::FT z);

introduces a vector v initialized to (x,y,z).

Operations

bool *v.operator==(w) const*

Test for equality: two vectors are equal, iff their x, y and z coordinates are equal. You can compare a vector with the *NULL_VECTOR*.

bool *v.operator!=(w) const*

Test for inequality. You can compare a vector with the *NULL_VECTOR*.

There are two sets of coordinate access functions, namely to the homogeneous and to the Cartesian coordinates. They can be used independently from the chosen kernel model.

Kernel::RT *v.hx() const*

returns the homogeneous x coordinate.

Kernel::RT *v.hy() const*

returns the homogeneous y coordinate.

Kernel::RT *v.hz() const*

returns the homogeneous z coordinate.

Kernel::RT *v.hw() const*

returns the homogenizing coordinate.

Note that you do not loose information with the homogeneous representation, because the *FieldNumberType* is a quotient.

Kernel::FT *v.x() const*

returns the x-coordinate of v, that is hx/hw.

Kernel::FT *v.y() const*

returns the y-coordinate of v, that is hy/hw.

Kernel::FT *v.z() const*

returns the z coordinate of v, that is hz/hw.

The following operations are for convenience and for compatibility with higher dimensional vectors. Again they come in a Cartesian and homogeneous flavor.

Kernel::RT *v.homogeneous(int i) const*

returns the i'th homogeneous coordinate of v, starting with 0.
Precondition: $0 \leq i \leq 3$.

Kernel::FT *v.cartesian(int i) const*

returns the i'th Cartesian coordinate of v, starting at 0.
Precondition: $0 \leq i \leq 2$.

Kernel::FT *v.operator[](int i) const*

returns *cartesian(i)*.
Precondition: $0 \leq i \leq 2$.

Cartesian_const_iterator *v.cartesian_begin() const*

returns an iterator to the Cartesian coordinates of v, starting with the 0th coordinate.

Cartesian_const_iterator *v.cartesian_end() const*

returns an off the end iterator to the Cartesian coordinates of v.

int *v.dimension() const*

returns the dimension (the constant 3).

Vector_3<Kernel> *v.transform(Aff_transformation_3<Kernel> t) const*

returns the vector obtained by applying t on v.

Direction_3<Kernel> *v.direction() const*

returns the direction of v.

Operators

The following operations can be applied on vectors:

Vector_3<Kernel> *v.operator+(w) const*

Addition.

Vector_3<Kernel> *v.operator-(w) const*

Subtraction.

Vector_3<Kernel> *v.operator-() const*

Returns the opposite vector.

Kernel::FT *v.operator*(w) const*

returns the scalar product (= inner product) of the two vectors.

Vector_3<Kernel> *operator*(v, Kernel::RT s)*

Multiplication with a scalar from the right.

Vector_3<Kernel> *operator*(v, Kernel::FT s)*

Multiplication with a scalar from the right.

Vector_3<Kernel> *operator*(Kernel::RT s, v)*

Multiplication with a scalar from the left.

Vector_3<Kernel> *operator*(Kernel::FT s, v)*

Multiplication with a scalar from the left.

Vector_3<Kernel> *v.operator/(Kernel::RT s) const*

Division by a scalar.

Kernel::FT *v.squared_length() const*

returns the squared length of *v*.

See Also

Kernel::Vector_3 . . . . page 807
CGAL::cross_product . . . . page 728
CGAL::determinant . . . . page 729

11.12 Constants and Enumerations

CGAL::Angle

#include <CGAL/enum.h>

enum Angle { *OBTUSE, RIGHT, ACUTE*};

Enum

See Also

CGAL::angle . . . . page 689

CGAL::Bounded_side

#include <CGAL/enum.h>

enum Bounded_side { *ON_UNBOUNDED_SIDE, ON_BOUNDARY, ON_BOUNDED_SIDE*};

Enum

Enum

CGAL::Comparison_result

#include <CGAL/enum.h>

enum Comparison_result { *SMALLER, EQUAL, LARGER*}*;*

Enum

CGAL::Sign

#include <CGAL/enum.h>

enum Sign { *NEGATIVE, ZERO, POSITIVE*}*;*

See Also

CGAL::Orientation . . . . page 679

CGAL::Orientation

#include <CGAL/enum.h>

typedef Sign Orientation;

See Also

CGAL::LEFT_TURN .... page 681
CGAL::RIGHT_TURN .... page 681
CGAL::COLLINEAR .... page 681
CGAL::CLOCKWISE .... page 680
CGAL::COUNTERCLOCKWISE .... page 680
CGAL::COPLANAR .... page 682

Enum

CGAL::Oriented_side

#include <CGAL/enum.h>

enum Oriented_side { *ON_NEGATIVE_SIDE, ON_ORIENTED_BOUNDARY, ON_POSITIVE_SIDE*}*;*

Enum

Constant

CGAL::CLOCKWISE

const Orientation *CLOCKWISE = NEGATIVE;*

See Also

CGAL::COUNTERCLOCKWISE . . . . page 680

Constant

CGAL::COUNTERCLOCKWISE

const Orientation *COUNTERCLOCKWISE = POSITIVE;*

See Also

CGAL::CLOCKWISE . . . . page 680

CGAL::COLLINEAR

const Orientation *COLLINEAR = ZERO;*

See Also

CGAL::LEFT_TURN . . . . page 681
CGAL::RIGHT_TURN . . . . page 681

Constant

CGAL::LEFT_TURN

const Orientation *LEFT_TURN = POSITIVE;*

See Also

CGAL::COLLINEAR . . . . page 681
CGAL::RIGHT_TURN . . . . page 681

Constant

CGAL::RIGHT_TURN

const Orientation *RIGHT_TURN = NEGATIVE;*

See Also

CGAL::COLLINEAR . . . . page 681 *CGAL::LEFT_TURN* . . . . page 681

Constant

Constant

CGAL::COPLANAR

const Orientation *COPLANAR = ZERO;*

Constant

CGAL::DEGENERATE

const Orientation *DEGENERATE = ZERO;*

CGAL::Null_vector

#include <CGAL/Origin.h>

Definition

CGAL defines a symbolic constant *NULL_VECTOR* to construct zero length vectors. *Null_vector* is the type of this constant.

See Also

CGAL::Vector_2<Kernel> . . . . page 635
CGAL::Vector_3<Kernel> . . . . page 673

Class

CGAL::NULL_VECTOR

const Null_vector *NULL_VECTOR;*

Definition

A symbolic constant used to construct zero length vectors.

See Also

CGAL::Vector_2<Kernel> . . . . page 635
CGAL::Vector_3<Kernel> . . . . page 673

Constant

Class

CGAL::Origin

#include <CGAL/Origin.h>

Definition

CGAL defines a symbolic constant *ORIGIN* which denotes the point at the origin. *Origin* is the type of this constant. It is used in the conversion between points and vectors.

See Also

CGAL::Point_2<Kernel> .... page 625
CGAL::Point_3<Kernel> .... page 655
CGAL::Vector_2<Kernel> .... page 635
CGAL::Vector_3<Kernel> .... page 673
CGAL::operator+ .... page 757
CGAL::operator- .... page 758

Constant

CGAL::ORIGIN

const Origin   *ORIGIN;*

Definition

A symbolic constant which denotes the point at the origin. This constant is used in the conversion between points and vectors.

Example

```
Point_2< Cartesian<Exact_NT> >  p(1.0, 1.0), q;
Vector2< Cartesian<Exact_NT> >  v;
v = p - ORIGIN;
q = ORIGIN + v;
assert( p == q );
```

See Also

CGAL::Point_2<Kernel> . . . . page 625
CGAL::Point_3<Kernel> . . . . page 655

Class

CGAL::Identity_transformation

#include <CGAL/aff_transformation_tags.h>

Definition

Tag class for affine transformations.

See Also

CGAL::Aff_transformation_2<Kernel> .... page 609
CGAL::Aff_transformation_3<Kernel> .... page 641
CGAL::Reflection .... page 686
CGAL::Rotation .... page 687
CGAL::Scaling .... page 687
CGAL::Translation .... page 688

Class

CGAL::Reflection

#include <CGAL/aff_transformation_tags.h>

Definition

Tag class for affine transformations.

See Also

CGAL::Aff_transformation_2<Kernel> .... page 609
CGAL::Aff_transformation_3<Kernel> .... page 641
CGAL::Identity_transformation .... page 686
CGAL::Rotation .... page 687
CGAL::Scaling .... page 687
CGAL::Translation .... page 688

CGAL::Rotation

#include <CGAL/aff_transformation_tags.h>

Definition

Tag class for affine transformations.

See Also

Class

CGAL::Scaling

#include <CGAL/aff_transformation_tags.h>

Definition

Tag class for affine transformations.

See Also

Class

Class

CGAL::Translation

#include <CGAL/aff_transformation_tags.h>

Definition

Tag class for affine transformations.

See Also

CGAL::Aff_transformation_2<Kernel> .... page 609
CGAL::Aff_transformation_3<Kernel> .... page 641
CGAL::Identity_transformation .... page 686
CGAL::Reflection .... page 686
CGAL::Rotation .... page 687
CGAL::Scaling .... page 687

11.13 Global Functions

CGAL::angle

Angle *angle(Vector_2<Kernel> u, Vector_2<Kernel> v)*

returns *OBTUSE*, *RIGHT* or *ACUTE* depending on the angle formed by the two vectors *u* and *v*.

Angle *angle(Point_2<Kernel> p, Point_2<Kernel> q, Point_2<Kernel> r)*

returns *OBTUSE*, *RIGHT* or *ACUTE* depending on the angle formed by the three points *p*, *q*, *r* (*q* being the vertex of the angle). The returned value is the same as *angle(p - q, r - q)*.

Angle *angle(Point_2<Kernel> p,*
Point_2<Kernel> q,
Point_2<Kernel> r,
Point_2<Kernel> s)

returns *OBTUSE*, *RIGHT* or *ACUTE* depending on the angle formed by the two vectors *pq*, *rs*. The returned value is the same as *angle(q - p, s - r)*.

Angle *angle(Point_3<Kernel> p, Point_3<Kernel> q, Point_3<Kernel> r)*

returns *OBTUSE*, *RIGHT* or *ACUTE* depending on the angle formed by the three points *p*, *q*, *r* (*q* being the vertex of the angle).

CGAL::are_ordered_along_line

bool	*are_ordered_along_line(*	*Point_2<Kernel> p, Point_2<Kernel> q, Point_2<Kernel> r)*

returns *true*, iff the three points are collinear and q lies between p and r. Note that *true* is returned, if $q{=}{=}p$ or $q{=}{=}r$.

bool	*are_ordered_along_line(*	*Point_3<Kernel> p, Point_3<Kernel> q, Point_3<Kernel> r)*

returns *true*, iff the three points are collinear and q lies between p and r. Note that *true* is returned, if $q{=}{=}p$ or $q{=}{=}r$.

See Also

CGAL::are_strictly_ordered_along_line . . . . page 691
CGAL::collinear_are_ordered_along_line . . . . page 701
CGAL::collinear_are_strictly_ordered_along_line . . . . page 702

CGAL::are_strictly_ordered_along_line

bool *are_strictly_ordered_along_line(Point_2<Kernel> p, Point_2<Kernel> q, Point_2<Kernel> r)*

returns *true*, iff the three points are collinear and q lies strictly between p and r. Note that *false* is returned, if $q==p$ or $q==r$.

bool *are_strictly_ordered_along_line(Point_3<Kernel> p, Point_3<Kernel> q, Point_3<Kernel> r)*

returns *true*, iff the three points are collinear and q lies strictly between p and r. Note that *false* is returned, if $q==p$ or $q==r$.

See Also

CGAL::are_ordered_along_line . . . . page 690
CGAL::collinear_are_ordered_along_line . . . . page 701
CGAL::collinear_are_strictly_ordered_along_line . . . . page 702

Function

Function

CGAL::area

Kernel::FT	*area(Point_2<Kernel> p, Point_2<Kernel> q, Point_2<Kernel> r)*	
		returns the signed area of the triangle defined by the points p, q and r.

CGAL::barycenter

Point_2<Kernel> *barycenter(Point_2<Kernel> p1, Kernel::FT w1, Point_2<Kernel> p2)*

compute the barycenter of the points $p1$ and $p2$ with corresponding weights $w1$ and $1 - w1$.

Point_2<Kernel> *barycenter(Point_2<Kernel> p1,*
Kernel::FT w1,
Point_2<Kernel> p2,
Kernel::FT w2)

compute the barycenter of the points $p1$ and $p2$ with corresponding weights $w1$ and $w2$.
Precondition: w1+w2 != 0.

Point_2<Kernel> *barycenter(Point_2<Kernel> p1,*
Kernel::FT w1,
Point_2<Kernel> p2,
Kernel::FT w2,
Point_2<Kernel> p3)

compute the barycenter of the points $p1$, $p2$ and $p3$ with corresponding weights $w1$, $w2$ and $1 - w1 - w2$.

Point_2<Kernel> *barycenter(Point_2<Kernel> p1,*
Kernel::FT w1,
Point_2<Kernel> p2,
Kernel::FT w2,
Point_2<Kernel> p3,
Kernel::FT w3)

compute the barycenter of the points $p1$, $p2$ and $p3$ with corresponding weights $w1$, $w2$ and $w3$.
Precondition: w1+w2+w3 != 0.

Point_2<Kernel> *barycenter(Point_2<Kernel> p1,*
Kernel::FT w1,
Point_2<Kernel> p2,
Kernel::FT w2,
Point_2<Kernel> p3,
Kernel::FT w3,
Point_2<Kernel> p4)

compute the barycenter of the points $p1$, $p2$, $p3$ and $p4$ with corresponding weights $w1$, $w2$, $w3$ and $1 - w1 - w2 - w3$.

Point_2<Kernel> *barycenter(Point_2<Kernel> p1,*
Kernel::FT w1,
Point_2<Kernel> p2,
Kernel::FT w2,
Point_2<Kernel> p3,
Kernel::FT w3,
Point_2<Kernel> p4,
Kernel::FT w4)

compute the barycenter of the points $p1$, $p2$, $p3$ and $p4$ with corresponding weights $w1$, $w2$, $w3$ and $w4$.
Precondition: w1+w2+w3+w4 != 0.

Point_3<Kernel> *barycenter(Point_3<Kernel> p1, Kernel::FT w1, Point_3<Kernel> p2)*

compute the barycenter of the points $p1$ and $p2$ with corresponding weights $w1$ and $1-w1$.

Point_3<Kernel> *barycenter(Point_3<Kernel> p1,*
Kernel::FT w1,
Point_3<Kernel> p2,
Kernel::FT w2)

compute the barycenter of the points $p1$ and $p2$ with corresponding weights $w1$ and $w2$.
Precondition: w1+w2 != 0.

Point_3<Kernel> *barycenter(Point_3<Kernel> p1,*
Kernel::FT w1,
Point_3<Kernel> p2,
Kernel::FT w2,
Point_3<Kernel> p3)

compute the barycenter of the points $p1$, $p2$ and $p3$ with corresponding weights $w1$, $w2$ and $1-w1-w2$.

Point_3<Kernel> *barycenter(Point_3<Kernel> p1,*
Kernel::FT w1,
Point_3<Kernel> p2,
Kernel::FT w2,
Point_3<Kernel> p3,
Kernel::FT w3)

compute the barycenter of the points $p1$, $p2$ and $p3$ with corresponding weights $w1$, $w2$ and $w3$.
Precondition: w1+w2+w3 != 0.

Point_3<Kernel> *barycenter(Point_3<Kernel> p1,*

Kernel::FT w1,
Point_3<Kernel> p2,
Kernel::FT w2,
Point_3<Kernel> p3,
Kernel::FT w3,
Point_3<Kernel> p4)

compute the barycenter of the points $p1$, $p2$, $p3$ and $p4$ with corresponding weights $w1$, $w2$, $w3$ and $1-w1-w2-w3$.

Point_3<Kernel> *barycenter(Point_3<Kernel> p1,*
Kernel::FT w1,
Point_3<Kernel> p2,
Kernel::FT w2,
Point_3<Kernel> p3,
Kernel::FT w3,
Point_3<Kernel> p4,
Kernel::FT w4)

compute the barycenter of the points $p1$, $p2$, $p3$ and $p4$ with corresponding weights $w1$, $w2$, $w3$ and $w4$.
Precondition: w1+w2+w3+w4 != 0.

See Also

CGAL::centroid . . . . page 3969

CGAL::bisector

Line_2<Kernel> *bisector(Point_2<Kernel> p, Point_2<Kernel> q)*

constructs the bisector line of the two points p and q. The bisector is oriented in such a way that p lies on its positive side.
Precondition: p and q are not equal.

Line_2<Kernel> *bisector(Line_2<Kernel> l1, Line_2<Kernel> l2)*

constructs the bisector of the two lines $l1$ and $l2$. In the general case, the bisector has the direction of the vector which is the sum of the normalized directions of the two lines, and which passes through the intersection of *l1* and *l2*. If *l1* and *l2* are parallel, then the bisector is defined as the line which has the same direction as *l1*, and which is at the same distance from *l1* and *l2*. This function requires that *Kernel::RT* supports the *sqrt()* operation.

Plane_3<Kernel> *bisector(Point_3<Kernel> p, Point_3<Kernel> q)*

constructs the bisector plane of the two points p and q. The bisector is oriented in such a way that p lies on its positive side.
Precondition: p and q are not equal.

Plane_3<Kernel> *bisector(Plane_3<Kernel> h1, Plane_3<Kernel> h2)*

constructs the bisector of the two planes $h1$ and $h2$. In the general case, the bisector has a normal vector which has the same direction as the sum of the normalized normal vectors of the two planes, and passes through the intersection of *h1* and *h2*. If *h1* and *h2* are parallel, then the bisector is defined as the plane which has the same oriented normal vector as *l1*, and which is at the same distance from *h1* and *h2*. This function requires that *Kernel::RT* supports the *sqrt()* operation.

CGAL::centroid

Function

Point_2<Kernel> *centroid(Point_2<Kernel> p, Point_2<Kernel> q, Point_2<Kernel> r)*

compute the centroid of the points p, q, and r.

Point_2<Kernel> *centroid(Point_2<Kernel> p, Point_2<Kernel> q, Point_2<Kernel> r, Point_2<Kernel> s)*

compute the centroid of the points p, q, r, and s.

Point_2<Kernel> *centroid(Triangle_2<Kernel> t)*

compute the centroid of the triangle t.

Point_3<Kernel> *centroid(Point_3<Kernel> p, Point_3<Kernel> q, Point_3<Kernel> r)*

compute the centroid of the points p, q, and r.

Point_3<Kernel> *centroid(Point_3<Kernel> p, Point_3<Kernel> q, Point_3<Kernel> r, Point_3<Kernel> s)*

compute the centroid of the points p, q, r, and s.

Point_3<Kernel> *centroid(Triangle_3<Kernel> t)*

compute the centroid of the triangle t.

Point_3<Kernel> *centroid(Tetrahedron_3<Kernel> t)*

compute the centroid of the tetrahedron t.

See Also

CGAL::barycenter . . . . page 3971

CGAL::circumcenter

Point_2<Kernel> *circumcenter(Point_2<Kernel> p, Point_2<Kernel> q)*

compute the center of the smallest circle passing through the points p and q. Note: this is the same as *CGAL::midpoint(p, q)* but is provided for homogeneity.

Point_2<Kernel> *circumcenter(Point_2<Kernel> p, Point_2<Kernel> q, Point_2<Kernel> r)*

compute the center of the circle passing through the points p, q, and r.
Precondition: p, q, and r are not collinear.

Point_2<Kernel> *circumcenter(Triangle_2<Kernel> t)*

compute the center of the circle passing through the vertices of t.
Precondition: t is not degenerate.

Point_3<Kernel> *circumcenter(Point_3<Kernel> p, Point_3<Kernel> q)*

compute the center of the smallest sphere passing through the points p and q. Note: this is the same as *CGAL::midpoint(p, q)* but is provided for homogeneity.

Point_3<Kernel> *circumcenter(Point_3<Kernel> p, Point_3<Kernel> q, Point_3<Kernel> r)*

compute the center of the circle passing through the points p, q, and r.
Precondition: p, q, and r are not collinear.

Point_3<Kernel> *circumcenter(Triangle_3<Kernel> t)*

compute the center of the circle passing through the vertices of t.
Precondition: t is not degenerate.

Point_3<Kernel> *circumcenter(Point_3<Kernel> p,*
Point_3<Kernel> q,
Point_3<Kernel> r,
Point_3<Kernel> s)

compute the center of the sphere passing through the points p, q, r, and s.
Precondition: p, q, r, and s are not coplanar.

Point_3<Kernel> *circumcenter(Tetrahedron_3<Kernel> t)*

compute the center of the sphere passing through the vertices of t.
Precondition: t is not degenerate.

Function

CGAL::collinear

bool *collinear(Point_2<Kernel> p, Point_2<Kernel> q, Point_2<Kernel> r)*

returns *true*, iff p, q, and r are collinear.

bool *collinear(Point_3<Kernel> p, Point_3<Kernel> q, Point_3<Kernel> r)*

returns *true*, iff p, q, and r are collinear.

See Also

CGAL::left_turn . . . . page 746
CGAL::orientation . . . . page 1116
CGAL::right_turn . . . . page 768

CGAL::collinear_are_ordered_along_line

Function

bool *collinear_are_ordered_along_line(Point_2<Kernel> p, Point_2<Kernel> q, Point_2<Kernel> r)*

returns *true*, iff q lies between p and r.
Precondition: p, q and r are collinear.

bool *collinear_are_ordered_along_line(Point_3<Kernel> p, Point_3<Kernel> q, Point_3<Kernel> r)*

returns *true*, iff q lies between p and r.
Precondition: p, q and r are collinear.

See Also

CGAL::are_ordered_along_line . . . page 690
CGAL::are_strictly_ordered_along_line . . . page 691
CGAL::collinear_are_strictly_ordered_along_line . . . page 702

Function

CGAL::collinear_are_strictly_ordered_along_line

bool	*collinear_are_strictly_ordered_along_line(Point_2<Kernel> p, Point_2<Kernel> q, Point_2<Kernel> r)*	
		returns *true*, iff q lies strictly between p and r. *Precondition*: p, q and r are collinear.
bool	*collinear_are_strictly_ordered_along_line(Point_3<Kernel> p, Point_3<Kernel> q, Point_3<Kernel> r)*	
		returns *true*, iff q lies strictly between p and r. *Precondition*: p, q and r are collinear.

See Also

CGAL::are_ordered_along_line . . . . page 690
CGAL::are_strictly_ordered_along_line . . . . page 691
CGAL::collinear_are_ordered_along_line . . . . page 701

CGAL::compare_distance_to_point

Function

Comparison_result *compare_distance_to_point(Point_2<Kernel> p, Point_2<Kernel> q, Point_2<Kernel> r)*

compares the distances of points q and r to point p. returns *SMALLER*, iff q is closer to p than r, *LARGER*, iff r is closer to p than q, and *EQUAL* otherwise.

Comparison_result *compare_distance_to_point(Point_3<Kernel> p, Point_3<Kernel> q, Point_3<Kernel> r)*

compares the distances of points q and r to point p. returns *SMALLER*, iff q is closer to p than r, *LARGER*, iff r is closer to p than q, and *EQUAL* otherwise.

See Also

CGAL::compare_squared_distance . . . . page 710
CGAL::compare_signed_distance_to_line . . . . page 705
CGAL::compare_signed_distance_to_plane . . . . page 706
CGAL::has_larger_distance_to_point . . . . page 735
CGAL::has_larger_signed_distance_to_line . . . . page 736
CGAL::has_larger_signed_distance_to_plane . . . . page 737
CGAL::has_smaller_distance_to_point . . . . page 738
CGAL::has_smaller_signed_distance_to_line . . . . page 739
CGAL::has_smaller_signed_distance_to_plane . . . . page 740

Function

CGAL::compare_lexicographically

Comparison_result *compare_lexicographically(Point_2<Kernel> p, Point_2<Kernel> q)*

Compares the Cartesian coordinates of points p and q lexicographically in xy order: first x-coordinates are compared, if they are equal, y-coordinates are compared. This is the same function as *compare_xy* and exists for compatibility with *Point_d<Kernel>*.

Comparison_result *compare_lexicographically(Point_3<Kernel> p, Point_3<Kernel> q)*

Compares the Cartesian coordinates of points p and q lexicographically in xyz order: first x-coordinates are compared, if they are equal, y-coordinates are compared, and if both x- and y- coordinate are equal, z-coordinates are compared. This is the same function as *compare_xyz* and exists for compatibility with *Point_d<Kernel>*.

CGAL::compare_signed_distance_to_line

Comparison_result *compare_signed_distance_to_line(Line_2<Kernel> l, Point_2<Kernel> p, Point_2<Kernel> q)*

returns *LARGER* iff the signed distance of p and l is larger than the signed distance of q and l, *SMALLER*, iff it is smaller, and *EQUAL* iff both are equal.

Comparison_result *compare_signed_distance_to_line(Point_2<Kernel> p, Point_2<Kernel> q, Point_2<Kernel> r, Point_2<Kernel> s)*

returns *LARGER* iff the signed distance of r and l is larger than the signed distance of s and l, *SMALLER*, iff it is smaller, and *EQUAL* iff both are equal, where l is the directed line through p and q.

See Also

CGAL::compare_distance_to_point . . . . page 703
CGAL::compare_signed_distance_to_plane . . . . page 706
CGAL::has_larger_distance_to_point . . . . page 735
CGAL::has_larger_signed_distance_to_line . . . . page 736
CGAL::has_larger_signed_distance_to_plane . . . . page 737
CGAL::has_smaller_distance_to_point . . . . page 738
CGAL::has_smaller_signed_distance_to_line . . . . page 739
CGAL::has_smaller_signed_distance_to_plane . . . . page 740

Function

CGAL::compare_signed_distance_to_plane

Comparison_result *compare_signed_distance_to_plane(Plane_3<Kernel> h, Point_3<Kernel> p, Point_3<Kernel> q)*

returns *LARGER* iff the signed distance of p and h is larger than the signed distance of q and h, *SMALLER*, iff it is smaller, and *EQUAL* iff both are equal.

Comparison_result *compare_signed_distance_to_plane(Point_3<Kernel> p, Point_3<Kernel> q, Point_3<Kernel> r, Point_3<Kernel> s, Point_3<Kernel> t)*

returns *LARGER* iff the signed distance of s and h is larger than the signed distance of t and h, *SMALLER*, iff it is smaller, and *EQUAL* iff both are equal, where h is the oriented plane through p, q and r.

See Also

CGAL::compare_distance_to_point . . . . . . page 703
CGAL::compare_signed_distance_to_line . . . . . . page 705
CGAL::has_larger_distance_to_point . . . . . . page 735
CGAL::has_larger_signed_distance_to_line . . . . . . page 736
CGAL::has_larger_signed_distance_to_plane . . . . . . page 737
CGAL::has_smaller_distance_to_point . . . . . . page 738
CGAL::has_smaller_signed_distance_to_line . . . . . . page 739
CGAL::has_smaller_signed_distance_to_plane . . . . . . page 740

CGAL::compare_slopes

Function

Comparison_result *compare_slopes(Line_2<Kernel> l1, Line_2<Kernel> l2)*

compares the slopes of the lines *l1* and *l2*

Comparison_result *compare_slopes(Segment_2<Kernel> s1, Segment_2<Kernel> s2)*

compares the slopes of the segments *s1* and *s2*

Function

CGAL::compare_dihedral_angle

Comparison_result *compare_dihedral_angle(Point_3<K> a1, Point_3<K> b1, Point_3<K> c1, Point_3<K> d1, K::FT cosine)*

compares the dihedral angles θ_1 and θ_2, where θ_1 is the dihedral angle, in $[0,\pi]$, of the tetrahedron (a_1,b_1,c_1,d_1) at the edge (a_1,b_1), and θ_2 is the angle in $[0,\pi]$ such that $cos(\theta_2) = cosine$. The result is the same as *compare_dihedral_angle(b1-a1, c1-a1, d1-a1, cosine)*.
Precondition: a_1, b_1, c_1 are not collinear, and a_1, b_1, d_1 are not collinear.

Comparison_result *compare_dihedral_angle(Point_3<K> a1, Point_3<K> b1, Point_3<K> c1, Point_3<K> d1, Point_3<K> a2, Point_3<K> b2, Point_3<K> c2, Point_3<K> d2)*

compares the dihedral angles θ_1 and θ_2, where θ_i is the dihedral angle in the tetrahedron (a_i,b_i,c_i,d_i) at the edge (a_i,b_i). These two angles are computed in $[0,\pi]$. The result is the same as *compare_dihedral_angle(b1-a1, c1-a1, d1-a1, b2-a2, c2-a2, d2-a2)*.
Precondition: For $i \in \{1,2\}$, a_i, b_i, c_i are not collinear, and a_i, b_i, d_i are not collinear.

Comparison_result *compare_dihedral_angle(K::Vector_3 u1, K::Vector_3 v1, K::Vector_3 w1, K::FT cosine)*

compares the dihedral angles θ_1 and θ_2, where θ_1 is the dihedral angle, in $[0,\pi]$, between the vectorial planes defined by (u_1,v_1) and (u_1,w_1), and θ_2 is the angle in $[0,\pi]$ such that $cos(\theta_2) = cosine$.
Precondition: u_1 and v_1 are not collinear, and u_1 and w_1 are not collinear.

Comparison_result *compare_dihedral_angle(K::Vector_3 u1, K::Vector_3 v1, K::Vector_3 w1, K::Vector_3 u2, K::Vector_3 v2,* *K::Vector_3 w2)*

compares the dihedral angles θ_1 and θ_2, where θ_i is the dihedral angle between the vectorial planes defined by (u_i, v_i) and (u_i, w_i). These two angles are computed in $[0, \pi]$.
Precondition: For $i \in \{1, 2\}$, u_i and v_i are not collinear, and u_i and w_i are not collinear.

Function

CGAL::compare_squared_distance

Comparison_result *compare_squared_distance(Point_2<Kernel> p, Point_2<Kernel> q, typename Kernel::FT d2)*

compares the squared distance of points p and q to $d2$.

Comparison_result *compare_squared_distance(Point_3<Kernel> p, Point_3<Kernel> q, typename Kernel::FT d2)*

compares the squared distance of points p and q to $d2$.

See Also

CGAL::compare_distance_to_point . . . . page 703
CGAL::compare_signed_distance_to_line . . . . page 705
CGAL::compare_signed_distance_to_plane . . . . page 706
CGAL::has_larger_distance_to_point . . . . page 735
CGAL::has_larger_signed_distance_to_line . . . . page 736
CGAL::has_larger_signed_distance_to_plane . . . . page 737
CGAL::has_smaller_distance_to_point . . . . page 738
CGAL::has_smaller_signed_distance_to_line . . . . page 739
CGAL::has_smaller_signed_distance_to_plane . . . . page 740

CGAL::compare_squared_radius

Comparison_result *compare_squared_radius(Point_3<Kernel> p, typename Kernel::FT sr)*

compares the squared radius of the sphere of radius 0 centered at p to sr. This returns the opposite sign of sr.

Comparison_result *compare_squared_radius(Point_3<Kernel> p, Point_3<Kernel> q, typename Kernel::FT sr)*

compares the squared radius of the sphere defined by the points p and q to sr.

Comparison_result *compare_squared_radius(Point_3<Kernel> p, Point_3<Kernel> q, Point_3<Kernel> r, typename Kernel::FT sr)*

compares the squared radius of the sphere defined by the points p, q, and r to sr.

Comparison_result *compare_squared_radius(Point_3<Kernel> p, Point_3<Kernel> q, Point_3<Kernel> r, Point_3<Kernel> s, typename Kernel::FT sr)*

compares the squared radius of the sphere defined by the points p, q, r, and r to sr.

CGAL::compare_x

Depending on which CGAL kernel is used, different versions of this global function are available. This is described below.

With the basic 2D and 3D Kernel (see Chapter 11)

Comparison_result *compare_x(Point_2<Kernel> p, Point_2<Kernel> q)*

compares the x-coordinates of p and q.

Comparison_result *compare_x(Point_3<Kernel> p, Point_3<Kernel> q)*

compares the x-coordinates of p and q.

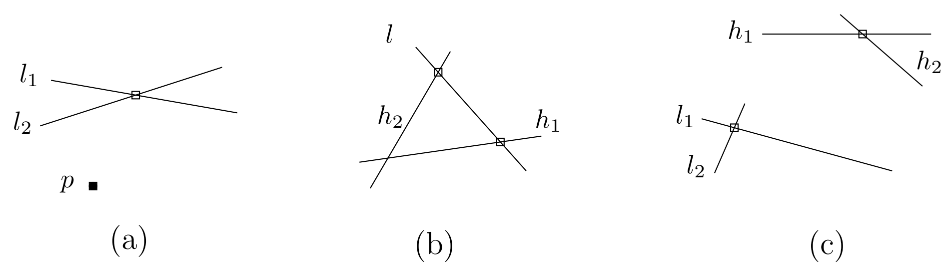


Figure 11.1: Comparison of the x or y-coordinates of the (implicitly given) points in the boxes.

Comparison_result *compare_x(Point_2<Kernel> p, Line_2<Kernel> l1, Line_2<Kernel> l2)*

compares the x-coordinates of p and the intersection of lines $l1$ and $l2$ (Figure 11.1 (a)).

Comparison_result *compare_x(Line_2<Kernel> l, Line_2<Kernel> h1, Line_2<Kernel> h2)*

compares the x-coordinates of the intersection of line l with line $h1$ and with line $h2$ (Figure 11.1 (b)).

Comparison_result *compare_x(Line_2<Kernel> l1,*
Line_2<Kernel> l2,
Line_2<Kernel> h1,
Line_2<Kernel> h2)

compares the x-coordinates of the intersection of lines $l1$ and $l2$ and the intersection of lines $h1$ and $h2$ (Figure 11.1 (c)).

With the 2D Circular Kernel (see Chapter 13)

#include <CGAL/global_functions_circular_kernel_2.h>

If this kernel is used, in addition to the function and the combination of 2D types described above, another version of the function is provided.

Comparison_result *compare_x(Circular_arc_point_2<CircularKernel> p, Circular_arc_point_2<CircularKernel> q)*

compares the x-coordinates of p and q.

Comparison_result *compare_x(Circular_arc_point_2<CircularKernel> p, Point_2<CircularKernel> q)*

compares the x-coordinates of p and q.

With the 3D Spherical Kernel (see Chapter 14)

#include <CGAL/global_functions_spherical_kernel_3.h>

If this kernel is used, in addition to the function and the combination of 2D types described above, another version of the function is provided.

Comparison_result *compare_x(Circular_arc_point_3<SphericalKernel> p, Circular_arc_point_3<SphericalKernel> q)*

compares the x-coordinates of p and q.

Comparison_result *compare_x(Circular_arc_point_3<SphericalKernel> p, Point_3<SphericalKernel> q)*

compares the x-coordinates of p and q.

See Also

CGAL::compare_xy

Depending on which CGAL kernel is used, different versions of this global function are available. This is described below.

With the basic 2D and 3D Kernel (see Chapter 11)

Comparison_result *compare_xy(Point_2<Kernel> p, Point_2<Kernel> q)*

Compares the Cartesian coordinates of points p and q lexicographically in xy order: first x-coordinates are compared, if they are equal, y-coordinates are compared.

Comparison_result *compare_xy(Point_3<Kernel> p, Point_3<Kernel> q)*

Compares the Cartesian coordinates of points p and q lexicographically in xy order: first x-coordinates are compared, if they are equal, y-coordinates are compared.

With the 2D Circular Kernel (see Chapter 13)

#include <CGAL/global_functions_circular_kernel_2.h>

If this kernel is used, in addition to the function and the combination of 2D types described above, another version of the function is provided.

Comparison_result *compare_xy(Circular_arc_point_2<CircularKernel> p, Circular_arc_point_2<CircularKernel> q)*

Compares the x and y Cartesian coordinates of points p and q lexicographically.

Comparison_result *compare_xy(Circular_arc_point_2<CircularKernel> p, Point_2<CircularKernel> q)*

Compares the x and y Cartesian coordinates of points p and q lexicographically.

With the 3D Spherical Kernel (see Chapter 14)

#include <CGAL/global_functions_spherical_kernel_3.h>

If this kernel is used, in addition to the function and the combination of 2D types described above, another version of the function is provided.

Comparison_result *compare_xy(Circular_arc_point_3<SphericalKernel> p, Circular_arc_point_3<SphericalKernel> q)*

Compares the x and y Cartesian coordinates of points p and q lexicographically.

Comparison_result *compare_xy(Circular_arc_point_3<SphericalKernel> p, Point_3<SphericalKernel> q)*

Compares the x and y Cartesian coordinates of points p and q lexicographically.

See Also

CGAL::compare_xyz . . . . . page 716
CGAL::compare_x . . . . . page 712
CGAL::compare_x_at_y . . . . . page 717
CGAL::compare_y . . . . . page 719
CGAL::compare_yx . . . . . page 723
CGAL::compare_y_at_x . . . . . page 721
CGAL::compare_z . . . . . page 724

Function

CGAL::compare_xyz

Depending on which CGAL kernel is used, different versions of this global function are available. This is described below.

With the basic 3D Kernel (see Chapter 11)

Comparison_result *compare_xyz(Point_3<Kernel> p, Point_3<Kernel> q)*

Compares the Cartesian coordinates of points p and q lexicographically in xyz order: first x-coordinates are compared, if they are equal, y-coordinates are compared, and if both x- and y- coordinate are equal, z-coordinates are compared.

With the 3D Spherical Kernel (see Chapter 14)

#include <CGAL/global_functions_spherical_kernel_3.h>

If this kernel is used, in addition to the function and the combination of 2D types described above, another version of the function is provided.

Comparison_result *compare_xyz(Circular_arc_point_3<SphericalKernel> p, Circular_arc_point_3<SphericalKernel> q)*

Compares the Cartesian coordinates of points p and q lexicographically.

Comparison_result *compare_xyz(Circular_arc_point_3<SphericalKernel> p, Point_3<SphericalKernel> q)*

Compares the Cartesian coordinates of points p and q lexicographically.

See Also

CGAL::compare_xy . . . . . . page 714
CGAL::compare_x . . . . . . page 712
CGAL::compare_x_at_y . . . . . . page 717
CGAL::compare_y . . . . . . page 719
CGAL::compare_yx . . . . . . page 723
CGAL::compare_y_at_x . . . . . . page 721
CGAL::compare_z . . . . . . page 724

CGAL::compare_x_at_y

Comparison_result *compare_x_at_y(Point_2<Kernel> p, Line_2<Kernel> h)*

compares the x-coordinates of p and the horizontal projection of p on h (Figure 11.2 (a)).
Precondition: h is not horizontal.

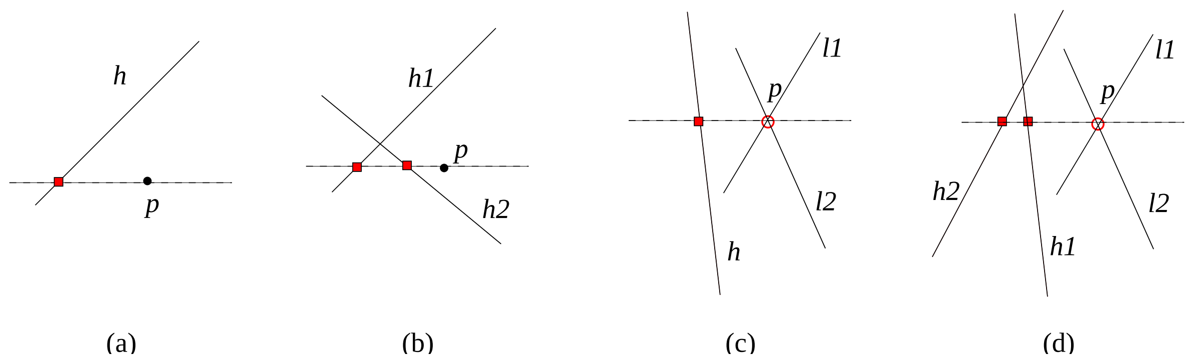


Figure 11.2: Comparison of the x-coordinates of the (implicitly given) points in the boxes, at a y-coordinate. The y-coordinate is either given explicitly (disc) or implicitly (circle).

Comparison_result *compare_x_at_y(Point_2<Kernel> p, Line_2<Kernel> h1, Line_2<Kernel> h2)*

This function compares the x-coordinates of the horizontal projection of p on *h1* and on *h2* (Figure 11.2 (b)).
Precondition: *h1* and *h2* are not horizontal.

Comparison_result *compare_x_at_y(Line_2<Kernel> l1, Line_2<Kernel> l2, Line_2<Kernel> h)*

Let p be the intersection of lines $l1$ and $l2$. This function compares the x-coordinates of p and the horizontal projection of p on h (Figure 11.2 (c)).
Precondition: *l1* and *l2* intersect and are not horizontal; h is not horizontal.

Comparison_result *compare_x_at_y(Line_2<Kernel> l1,*
Line_2<Kernel> l2,
Line_2<Kernel> h1,
Line_2<Kernel> h2)

Let p be the intersection of lines $l1$ and $l2$. This function compares the x-coordinates of the horizontal projection of p on *h1* and on *h2* (Figure 11.2 (d)).
Precondition: *l1* and *l2* intersect and are not horizontal; *h1* and *h2* are not horizontal.

See Also

CGAL::compare_xy . . . . page 714
CGAL::compare_xyz . . . . page 716
CGAL::compare_x . . . . page 712
CGAL::compare_y . . . . page 719
CGAL::compare_yx . . . . page 723
CGAL::compare_y_at_x . . . . page 721
CGAL::compare_z . . . . page 724

CGAL::compare_y

Depending on which CGAL kernel is used, different versions of this global function are available. This is described below.

With the basic 2D and 3D Kernel (see Chapter 11)

Comparison_result *compare_y(Point_2<Kernel> p, Point_2<Kernel> q)*

compares Cartesian y-coordinates of p and q.

Comparison_result *compare_y(Point_3<Kernel> p, Point_3<Kernel> q)*

compares Cartesian y-coordinates of p and q.

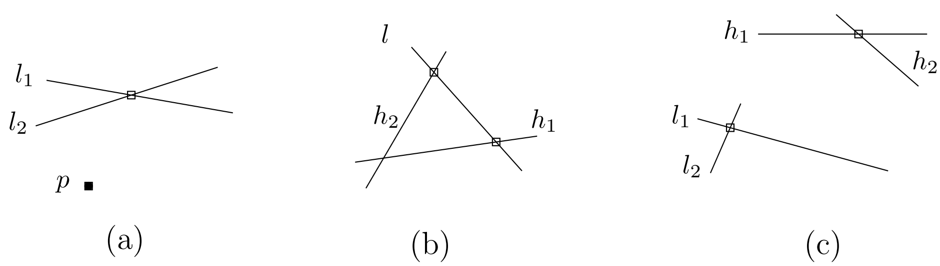


Figure 11.3: Comparison of the x or y-coordinates of the (implicitly given) points in the boxes.

Comparison_result *compare_y(Point_2<Kernel> p, Line_2<Kernel> l1, Line_2<Kernel> l2)*

compares the y-coordinates of p and the intersection of lines $l1$ and $l2$ (Figure 11.3 (a)).

Comparison_result *compare_y(Line_2<Kernel> l, Line_2<Kernel> h1, Line_2<Kernel> h2)*

compares the y-coordinates of the intersection of line l with line $h1$ and with line $h2$ (Figure 11.3 (b)).

Comparison_result *compare_y(Line_2<Kernel> l1,*
Line_2<Kernel> l2,
Line_2<Kernel> h1,
Line_2<Kernel> h2)

compares the y-coordinates of the intersection of lines $l1$ and $l2$ and the intersection of lines $h1$ and $h2$ (Figure 11.3 (c)).

With the 2D Circular Kernel (see Chapter 13)

#include <CGAL/global_functions_circular_kernel_2.h>

If this kernel is used, in addition to the function and the combination of 2D types described above, another version of the function is provided.

Comparison_result *compare_y(Circular_arc_point_2<CircularKernel> p, Circular_arc_point_2<CircularKernel> q)*

compares the y-coordinates of p and q.

Comparison_result *compare_y(Circular_arc_point_2<CircularKernel> p, Point_2<CircularKernel> q)*

compares the y-coordinates of p and q.

With the 3D Spherical Kernel (see Chapter 14)

#include <CGAL/global_functions_spherical_kernel_3.h>

If this kernel is used, in addition to the function and the combination of 2D types described above, another version of the function is provided.

Comparison_result *compare_y(Circular_arc_point_3<SphericalKernel> p, Circular_arc_point_3<SphericalKernel> q)*

compares the y-coordinates of p and q.

Comparison_result *compare_y(Circular_arc_point_3<SphericalKernel> p, Point_3<SphericalKernel> q)*

compares the y-coordinates of p and q.

CGAL::compare_xy . . . . page 714
CGAL::compare_xyz . . . . page 716
CGAL::compare_x . . . . page 712
CGAL::compare_x_at_y . . . . page 717
CGAL::compare_yx . . . . page 723
CGAL::compare_y_at_x . . . . page 721
CGAL::compare_z . . . . page 724

CGAL::compare_y_at_x

Depending on which CGAL kernel is used, different versions of this global function are available. This is described below.

With the basic 2D and 3D Kernel (see Chapter 11)

Comparison_result *compare_y_at_x(Point_2<Kernel> p, Line_2<Kernel> h)*

compares the y-coordinates of p and the vertical projection of p on h (Figure 11.4 (d)).
Precondition: h is not vertical.

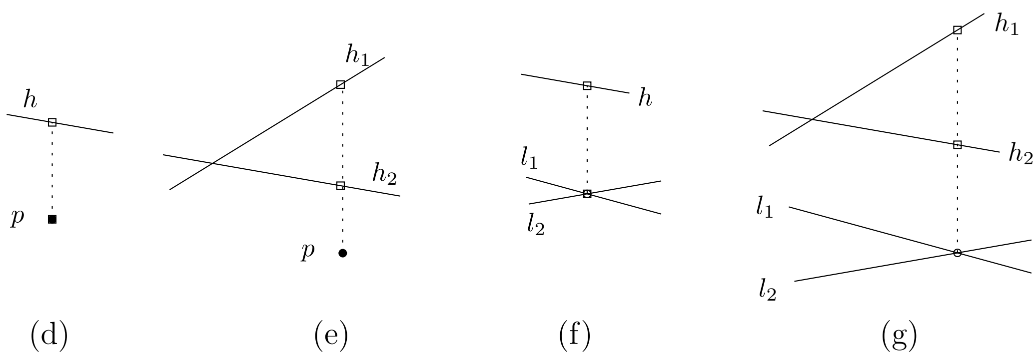


Figure 11.4: Comparison of the y-coordinates of the (implicitly given) points in the boxes, at an x-coordinate. The x-coordinate is either given explicitly (disc) or implicitly (circle).

Comparison_result *compare_y_at_x(Point_2<Kernel> p, Line_2<Kernel> h1, Line_2<Kernel> h2)*

compares the y-coordinates of the vertical projection of p on *h1* and on *h2* (Figure 11.4 (e)).
Precondition: *h1* and *h2* are not vertical.

Comparison_result *compare_y_at_x(Line_2<Kernel> l1, Line_2<Kernel> l2, Line_2<Kernel> h)*

Let p be the intersection of lines $l1$ and $l2$. This function compares the y-coordinates of p and the vertical projection of p on h (Figure 11.4 (f)).
Precondition: *l1*, *l2* intersect and h is not vertical.

Comparison_result *compare_y_at_x(Line_2<Kernel> l1,*
Line_2<Kernel> l2,
Line_2<Kernel> h1,
Line_2<Kernel> h2)

Let p be the intersection of lines $l1$ and $l2$. This function compares the y-coordinates of the vertical projection of p on *h1* and on *h2* (Figure 11.4 (g)).
Precondition: *l1* and *l2* intersect; *h1* and *h2* are not vertical.

Comparison_result *compare_y_at_x(Point_2<Kernel> p, Segment_2<Kernel> s)*

compares the y-coordinates of p and the vertical projection of p on s. If s is vertical, then return *EQUAL* when p lies on s, *SMALLER* when p lies under s, and *LARGER* otherwise.
Precondition: p is within the x range of s.

Comparison_result *compare_y_at_x(Point_2<Kernel> p, Segment_2<Kernel> s1, Segment_2<Kernel> s2)*

compares the y-coordinates of the vertical projection of p on $s1$ and on $s2$. If $s1$ or $s2$ is vertical, then return *EQUAL* if they intersect, otherwise return *SMALLER* if $s1$ lies below $s2$, and return *LARGER* otherwise.
Precondition: p is within the x range of $s1$ and $s2$.

With the 2D Circular Kernel (see Chapter 13)

#include <CGAL/global_functions_circular_kernel_2.h>

If this kernel is used, in addition to the function and the combination of 2D types described above, another version of the function is provided.

Comparison_result *compare_y_at_x(Circular_arc_point_2<CircularKernel> p, Circular_arc_2<CircularKernel> a)*

Same as above, for a point and a circular arc.

Comparison_result *compare_y_at_x(Circular_arc_point_2<CircularKernel> p, Line_arc_2<CircularKernel> a)*

Same as above, for a point and a line segment.

See Also

CGAL::compare_xy . . . . page 714
CGAL::compare_xyz . . . . page 716
CGAL::compare_x . . . . page 712
CGAL::compare_x_at_y . . . . page 717
CGAL::compare_y . . . . page 719
CGAL::compare_yx . . . . page 723
CGAL::compare_z . . . . page 724

CGAL::compare_yx

Comparison_result *compare_yx(Point_2<Kernel> p, Point_2<Kernel> q)*

Compares the Cartesian coordinates of points p and q lexicographically in yx order: first y-coordinates are compared, if they are equal, x-coordinates are compared.

See Also

CGAL::compare_xy . . . . . . page 714
CGAL::compare_xyz . . . . . . page 716
CGAL::compare_x . . . . . . page 712
CGAL::compare_x_at_y . . . . . . page 717
CGAL::compare_y . . . . . . page 719
CGAL::compare_y_at_x . . . . . . page 721
CGAL::compare_z . . . . . . page 724

Function

CGAL::compare_z

Depending on which CGAL kernel is used, different versions of this global function are available. This is described below.

With the basic 3D Kernel (see Chapter 11)

Comparison_result *compare_z(Point_3<Kernel> p, Point_3<Kernel> q)*

compares the z-coordinates of p and q.

With the 3D Spherical Kernel (see Chapter 14)

#include <CGAL/global_functions_spherical_kernel_3.h>

If this kernel is used, in addition to the function and the combination of 2D types described above, another version of the function is provided.

Comparison_result *compare_z(Circular_arc_point_3<SphericalKernel> p, Circular_arc_point_3<SphericalKernel> q)*

compares the z-coordinates of p and q.

Comparison_result *compare_z(Circular_arc_point_3<SphericalKernel> p, Point_3<SphericalKernel> q)*

compares the z-coordinates of p and q.

See Also

CGAL::coplanar

Function

bool *coplanar(Point_3<Kernel> p,*
Point_3<Kernel> q,
Point_3<Kernel> r,
Point_3<Kernel> s)

returns *true*, if p, q, r, and s are coplanar.

See Also

CGAL::coplanar_orientation . . . . . . page 726
CGAL::coplanar_side_of_bounded_circle . . . . . . page 727

Function

CGAL::coplanar_orientation

Orientation *coplanar_orientation(Point_3<Kernel> p, Point_3<Kernel> q, Point_3<Kernel> r, Point_3<Kernel> s)*

Let P be the plane defined by the points p, q, and r. Note that the order defines the orientation of P. The function computes the orientation of points p, q, and s in P: Iff p, q, s are collinear, *COLLINEAR* is returned. Iff P and the plane defined by p, q, and s have the same orientation, *POSITIVE* is returned; otherwise *NEGATIVE* is returned. *Precondition*: p, q, r, and s are coplanar and p, q, and r are not collinear.

Orientation *coplanar_orientation(Point_3<Kernel> p, Point_3<Kernel> q, Point_3<Kernel> r)*

If p,q,r are collinear, then *COLLINEAR* is returned. If not, then p,q,r define a plane P. The return value in this case is either *POSITIVE* or *NEGATIVE*, but we don't specify it explicitly. However, we guarantee that all calls to this predicate over 3 points in P will return a coherent orientation if considered a 2D orientation in P.

See Also

CGAL::coplanar . . . page 725
CGAL::coplanar_side_of_bounded_circle . . . page 727
CGAL::orientation . . . page 1116

Function

CGAL::coplanar_side_of_bounded_circle

Bounded_side *coplanar_side_of_bounded_circle(Point_3<Kernel> p, Point_3<Kernel> q, Point_3<Kernel> r, Point_3<Kernel> s)*

returns the bounded side of the circle defined by p, q, and r on which s lies.
Precondition: p, q, r, and s are coplanar and p, q, and r are not collinear.

See Also

CGAL::coplanar_orientation . . . . page 726
CGAL::side_of_bounded_circle . . . . page 769

Function

CGAL::cross_product

Vector_3<Kernel> *cross_product(Vector_3<Kernel> u, Vector_3<Kernel> v)*

returns the cross product of u and v.

CGAL::determinant

Kernel::FT *determinant(Vector_2<Kernel> v, Vector_2<Kernel> w)*

returns the determinant of v and w.

Kernel::FT *determinant(Vector_3<Kernel> u, Vector_3<Kernel> v, Vector_3<Kernel> w)*

returns the determinant of u, v and w.

See Also

CGAL::orientation . . . . page 1116
CGAL::collinear . . . . page 700
CGAL::left_turn . . . . page 746
CGAL::right_turn . . . . page 768

Function

CGAL::do_intersect

bool *do_intersect(Type1 obj1, Type2 obj2)*

checks whether *obj1* and *obj2* intersect. Two objects *obj1* and *obj2* intersect if there is a point *p* that is part of both *obj1* and *obj2*. The intersection region of those two objects is defined as the set of all points *p* that are part of both *obj1* and *obj2*. Note that for objects like triangles and polygons that enclose a bounded region, this region is part of the object.

Depending on which CGAL kernel is used, *Type1* and *Type2* can be of different types.

With the basic 2D and 3D Kernel (see Chapter 11)

#include <CGAL/intersections.h>

The types *Type1* and *Type2* can be any of the following:

- *Point_2<Kernel>*
- *Line_2<Kernel>*
- *Ray_2<Kernel>*
- *Segment_2<Kernel>*
- *Triangle_2<Kernel>*
- *Iso_rectangle_2<Kernel>*

Also, *Type1* and *Type2* can be both of type

- *Line_2<Kernel>*
- *Circle_2<Kernel>*

In three-dimensional space, the types *Type1* and *Type2* can be any of the following:

- *Plane_3<Kernel>*
- *Line_3<Kernel>*
- *Ray_3<Kernel>*
- *Segment_3<Kernel>*
- *Triangle_3<Kernel>*.
- *Bbox_3*.

Also, *Type1* and *Type2* can be respectively of types

- *Triangle_3<Kernel>* and *Tetrahedron_3<Kernel>*
- *Plane_3<Kernel>* and *Sphere_3<Kernel>* (or the contrary)
- *Sphere_3<Kernel>* and *Sphere_3<Kernel>*.

With the 2D Circular Kernel (see Chapter 13)

#include <CGAL/Circular_kernel_intersections.h>

If this kernel is used, in addition to the combinations of 2D types previously listed, *Type1* and *Type2* can be any of the following:

- *Line_2<CircularKernel>*
- *Circle_2<CircularKernel>*
- *Line_arc_2<CircularKernel>*
- *Circular_arc_2<CircularKernel>*

An example illustrating this is presented in Chapter 13.

With the 3D Spherical Kernel (see Chapter 14)

#include <CGAL/Spherical_kernel_intersections.h>

If this kernel is used, in addition to the combinations of 3D types previously listed, *Type1* and *Type2* can be any of the following:

- *Line_3<SphericalKernel>*
- *Circle_3<SphericalKernel>*
- *Plane_3<SphericalKernel>*
- *Sphere_3<SphericalKernel>*
- *Line_arc_3<SphericalKernel>*
- *Circular_arc_3<SphericalKernel>*

An example illustrating this is presented in Chapter 14.

Also, another predicate is provided with this kernel:

bool *do_intersect(Type1 obj1, Type2 obj2, Type3 obj3)*

checks whether *obj1*, *obj2* and *obj3* intersect.

where *Type1*, *Type2* and *Type3* can be:

- *Sphere_3<SphericalKernel>*
- *Plane_3<SphericalKernel>*

See Also

CGAL::intersection . . . . page 1553

Function

CGAL::do_overlap

#include <CGAL/Bbox_2.h>

bool *do_overlap(Bbox_2 bb1, Bbox_2 bb2)*

returns *true* iff *bb1* and *bb2* overlap, i.e., iff their intersection is non-empty.

#include <CGAL/Bbox_3.h>

bool *do_overlap(Bbox_3 bb1, Bbox_3 bb2)*

returns *true* iff *bb1* and *bb2* overlap, i.e., iff their intersection is non-empty.

CGAL::enum_cast

Function

#include <CGAL/enum.h>

template < typename T, typename U >		
T	*enum_cast(U u)*	converts between the various enums provided by the CGAL kernel. The conversion preserves the order of the values.

See Also

CGAL::Sign . . . . page 678
CGAL::Comparison_result . . . . page 678
CGAL::Orientation . . . . page 679
CGAL::Oriented_side . . . . page 679
CGAL::Bounded_side . . . . page 677
CGAL::Angle . . . . page 677
CGAL::Uncertain<T> . . . . page 4236

Function

CGAL::equidistant_line

Line_3<Kernel> *equidistant_line(Point_3<Kernel> p, Point_3<Kernel> q, Point_3<Kernel> r)*

constructs the line which is at the same distance from the three points p, q and r.
Precondition: p, q and r are not collinear.

CGAL::has_larger_distance_to_point

Function

bool *has_larger_distance_to_point(Point_2<Kernel> p, Point_2<Kernel> q, Point_2<Kernel> r)*

returns *true* iff the distance between q and p is larger than the distance between r and p.

bool *has_larger_distance_to_point(Point_3<Kernel> p, Point_3<Kernel> q, Point_3<Kernel> r)*

returns *true* iff the distance between q and p is larger than the distance between r and p.

See Also

CGAL::compare_distance_to_point . . . . . . page 703
CGAL::compare_signed_distance_to_line . . . . . . page 705
CGAL::compare_signed_distance_to_plane . . . . . . page 706
CGAL::has_larger_signed_distance_to_line . . . . . . page 736
CGAL::has_larger_signed_distance_to_plane . . . . . . page 737
CGAL::has_smaller_distance_to_point . . . . . . page 738
CGAL::has_smaller_signed_distance_to_line . . . . . . page 739
CGAL::has_smaller_signed_distance_to_plane . . . . . . page 740

Function

CGAL::has_larger_signed_distance_to_line

bool *has_larger_signed_distance_to_line(Line_2<Kernel> l, Point_2<Kernel> p, Point_2<Kernel> q)*

returns *true* iff the signed distance of p and l is larger than the signed distance of q and l.

bool *has_larger_signed_distance_to_line(Point_2<Kernel> p, Point_2<Kernel> q, Point_2<Kernel> r, Point_2<Kernel> s)*

returns *true* iff the signed distance of r and l is larger than the signed distance of s and l, where l is the directed line through points p and q.

See Also

CGAL::compare_distance_to_point ... page 703
CGAL::compare_signed_distance_to_line ... page 705
CGAL::compare_signed_distance_to_plane ... page 706
CGAL::has_larger_distance_to_point ... page 735
CGAL::has_larger_signed_distance_to_line ... page 736
CGAL::has_larger_signed_distance_to_plane ... page 737
CGAL::has_smaller_distance_to_point ... page 738
CGAL::has_smaller_signed_distance_to_line ... page 739
CGAL::has_smaller_signed_distance_to_plane ... page 740

CGAL::has_larger_signed_distance_to_plane

bool *has_larger_signed_distance_to_plane(Plane_3<Kernel> h, Point_3<Kernel> p, Point_3<Kernel> q)*

returns *true* iff the signed distance of p and h is larger than the signed distance of q and h.

bool *has_larger_signed_distance_to_plane(Point_3<Kernel> p, Point_3<Kernel> q, Point_3<Kernel> r, Point_3<Kernel> s, Point_3<Kernel> t)*

returns *true* iff the signed distance of s and h is larger than the signed distance of t and h, where h is the oriented plane through p, q and r.

See Also

CGAL::compare_distance_to_point . . . . . page 703
CGAL::compare_signed_distance_to_line . . . . . page 705
CGAL::compare_signed_distance_to_plane . . . . . page 706
CGAL::has_larger_distance_to_point . . . . . page 735
CGAL::has_larger_signed_distance_to_line . . . . . page 736
CGAL::has_smaller_distance_to_point . . . . . page 738
CGAL::has_smaller_signed_distance_to_line . . . . . page 739
CGAL::has_smaller_signed_distance_to_plane . . . . . page 740

Function

CGAL::has_smaller_distance_to_point

bool *has_smaller_distance_to_point(Point_2<Kernel> p, Point_2<Kernel> q, Point_2<Kernel> r)*

returns *true* iff the distance between q and p is smaller than the distance between r and p.

bool *has_smaller_distance_to_point(Point_3<Kernel> p, Point_3<Kernel> q, Point_3<Kernel> r)*

returns *true* iff the distance between q and p is smaller than the distance between r and p.

See Also

CGAL::compare_distance_to_point . . . . page 703
CGAL::compare_signed_distance_to_line . . . . page 705
CGAL::compare_signed_distance_to_plane . . . . page 706
CGAL::has_larger_distance_to_point . . . . page 735
CGAL::has_larger_signed_distance_to_line . . . . page 736
CGAL::has_larger_signed_distance_to_plane . . . . page 737
CGAL::has_smaller_signed_distance_to_line . . . . page 739
CGAL::has_smaller_signed_distance_to_plane . . . . page 740

CGAL::has_smaller_signed_distance_to_line

bool	*has_smaller_signed_distance_to_line(*	*Line_2<Kernel> l, Point_2<Kernel> p, Point_2<Kernel> q)*

returns *true* iff the signed distance of p and l is smaller than the signed distance of q and l.

bool	*has_smaller_signed_distance_to_line(*	*Point_2<Kernel> p, Point_2<Kernel> q, Point_2<Kernel> r, Point_2<Kernel> s)*

returns *true* iff the signed distance of r and l is smaller than the signed distance of s and l, where l is the oriented line through p and q.

See Also

CGAL::compare_distance_to_point . . . . page 703
CGAL::compare_signed_distance_to_line . . . . page 705
CGAL::compare_signed_distance_to_plane . . . . page 706
CGAL::has_larger_distance_to_point . . . . page 735
CGAL::has_larger_signed_distance_to_line . . . . page 736
CGAL::has_larger_signed_distance_to_plane . . . . page 737
CGAL::has_smaller_distance_to_point . . . . page 738
CGAL::has_smaller_signed_distance_to_plane . . . . page 740

Function

Function

CGAL::has_smaller_signed_distance_to_plane

bool *has_smaller_signed_distance_to_plane(Plane_3<Kernel> h, Point_3<Kernel> p, Point_3<Kernel> q)*

returns *true* iff the signed distance of p and h is smaller than the signed distance of q and h.

bool *has_smaller_signed_distance_to_plane(Point_3<Kernel> p, Point_3<Kernel> q, Point_3<Kernel> r, Point_3<Kernel> s, Point_3<Kernel> t)*

returns *true* iff the signed distance of p and h is smaller than the signed distance of q and h, where h is the oriented plane through p, q and r.

See Also

CGAL::compare_distance_to_point . . . . page 703
CGAL::compare_signed_distance_to_line . . . . page 705
CGAL::compare_signed_distance_to_plane . . . . page 706
CGAL::has_larger_distance_to_point . . . . page 735
CGAL::has_larger_signed_distance_to_line . . . . page 736
CGAL::has_larger_signed_distance_to_plane . . . . page 737
CGAL::has_smaller_distance_to_point . . . . page 738
CGAL::has_smaller_signed_distance_to_line . . . . page 739

CGAL::intersection

Depending on which CGAL kernel is used, different versions of this global function are available. This is described below.

With the basic 2D and 3D Kernel (see Chapter 11)

#include <CGAL/intersections.h>

Object *intersection(Type1<Kernel> obj1, Type2<Kernel> obj2)*

Two objects *obj1* and *obj2* intersect if there is a point *p* that is part of both *obj1* and *obj2*. The intersection region of those two objects is defined as the set of all points *p* that are part of both *obj1* and *obj2*. Note that for objects like triangles and polygons that enclose a bounded region, this region is considered part of the object. If a segment lies completely inside a triangle, then those two objects intersect and the intersection region is the complete segment.

The possible value for types *Type1* and *Type2* and the possible return values wrapped in *Object* are the following:

type A	type B	return type
Iso_rectangle_2	*Iso_rectangle_2*	*Iso_rectangle_2*
Iso_rectangle_2	*Line_2*	*Point_2* *Segment_2*
Iso_rectangle_2	*Ray_2*	*Point_2* *Segment_2*
Iso_rectangle_2	*Segment_2*	*Point_2* *Segment_2*
Iso_rectangle_2	*Triangle_2*	*Point_2* *Segment_2* *Triangle_2* *std::vector<Point_2>*
Line_2	*Line_2*	*Point_2* *Line_2*
Line_2	*Ray_2*	*Point_2* *Ray_2*
Line_2	*Segment_2*	*Point_2* *Segment_2*
Line_2	*Triangle_2*	*Point_2* *Segment_2*
Ray_2	*Ray_2*	*Point_2* *Segment_2* *Ray_2*

continued

Ray_2	*Segment_2*	*Point_2* *Segment_2*
Ray_2	*Triangle_2*	*Point_2* *Segment_2*
Segment_2	*Segment_2*	*Point_2* *Segment_2*
Segment_2	*Triangle_2*	*Point_2* *Segment_2*
Triangle_2	*Triangle_2*	*Point_2* *Segment_2* *Triangle_2* *std::vector<Point_2>*
Line_3	*Line_3*	*Point_3* *Line_3*
Line_3	*Plane_3*	*Point_3* *Line_3*
Line_3	*Ray_3*	*Point_3* *Ray_3*
Line_3	*Segment_3*	*Point_3* *Segment_3*
Line_3	*Triangle_3*	*Point_3* *Segment_3*
Plane_3	*Plane_3*	*Line_3* *Plane_3*
Plane_3	*Ray_3*	*Point_3* *Ray_3*
Plane_3	*Segment_3*	*Point_3* *Segment_3*
Plane_3	*Sphere_3*	*Point_3* *Circle_3*
Plane_3	*Triangle_3*	*Triangle_3* *Segment_3* *Point_3*
Ray_3	*Ray_3*	*Point_3* *Ray_3* *Segment_3*
Ray_3	*Segment_3*	*Point_3* *Segment_3*
Ray_3	*Triangle_3*	*Point_3* *Segment_3*
Segment_3	*Segment_3*	*Point_3* *Segment_3*
Segment_3	*Triangle_3*	*Point_3* *Segment_3*
Sphere_3	*Sphere_3*	*Point_3* *Circle_3* *Sphere_3*

continued

Triangle_3	*Triangle_3*	*Point_3* *Segment_3* *Triangle_3* *std::vector<Point_3>*

There is also an intersection function between 3 planes.

Object *intersection(Plane_3<Kernel> pl1, Plane_3<Kernel> pl2, Plane_3<Kernel> pl3)*

returns the intersection of 3 planes, which can be either a point, a line, a plane, or empty.

With the 2D Circular Kernel (see Chapter 13)

#include <CGAL/Circular_kernel_intersections.h>

If this kernel is used, in addition to the function and the combination of 2D types described above, another version of the function is provided.

Since both the number of intersections, if any, and their type, depend on the arguments, the function returns an output iterator on *Object*'s, as presented below.

template < class OutputIterator >
OutputIterator *intersection(Type1 obj1, Type2 obj2, OutputIterator intersections)*

Copies in the output iterator the intersection elements between the two objects. *intersections* iterates on elements of type *CGAL::Object*, in lexicographic order,

where *Type1* and *Type2* can both be either

- *Line_2<CircularKernel>* or
- *Line_arc_2<CircularKernel>* or
- *Circle_2<CircularKernel>* or
- *Circular_arc_2<CircularKernel>*.

Depending on the types *Type1* and *Type2*, these elements can be assigned to

- *std::pair<Circular_arc_point_2<CircularKernel>, unsigned>*, where the unsigned integer is the multiplicity of the corresponding intersection point between *obj1* and *obj2*,
- *Circular_arc_2<CircularKernel>* in case of an overlap of two circular arcs,
- *Line_arc_2<CircularKernel>* in case of an overlap of two line segments or
- *Line_2<CircularKernel>* or *Circle_2<CircularKernel>* in case of two equal input lines or circles.

With the 3D Spherical Kernel (see Chapter 14)

#include <CGAL/Spherical_kernel_intersections.h>

If this kernel is used, in addition to the function and the combination of 3D types described above, two other versions of the function are provided.

Since both the number of intersections, if any, and their type, depend on the arguments, the functions return an output iterator on *Object*'s, as presented below.

The **first function** is:

template < class OutputIterator >
OutputIterator *intersection(Type1 obj1, Type2 obj2, OutputIterator intersections)*

Copies in the output iterator the intersection elements between the two objects. *intersections* iterates on elements of type *CGAL::Object*, in lexicographic order, when this ordering is defined on the computed objects,

where *Type1* and *Type2* can both be either:

- *Sphere_3<SphericalKernel>*,
- *Plane_3<SphericalKernel>*,
- *Line_3<SphericalKernel>*,
- *Circle_3<SphericalKernel>*,
- *Line_arc_3<SphericalKernel>* or
- *Circular_arc_3<SphericalKernel>*,

and depending on the types *Type1* and *Type2*, the computed *CGAL::Object*s can be assigned to

- *std::pair<Circular_arc_point_3<SphericalKernel>, unsigned>*, where the unsigned integer is the multiplicity of the corresponding intersection point between *obj1* and *obj2*,
- *Type1*, when *Type1* and *Type2* are equal, and if the two objets *obj1* and *obj2* are equal,
- *Line_3<SphericalKernel>* or *Circle_3<SphericalKernel>* when *Type1* and *Type2* are two-dimensional objets intersecting along a curve (2 planes, or 2 spheres, or one plane and one sphere),
- *Circular_arc_3<SphericalKernel>* in case of an overlap of two circular arcs or
- *Line_arc_3<SphericalKernel>* in case of an overlap of two line segments.

The **second function** is:

template < class OutputIterator >
OutputIterator *intersection(Type1 obj1, Type2 obj2, Type3 obj3, OutputIterator intersections)*

Copies in the output iterator the intersection elements between the three objects. *intersections* iterates on elements of type *CGAL::Object*, in lexicographic order when this ordering is defined on the computed objects where *Type1*, *Type2* and *Type3* can be either

- *Sphere_3<SphericalKernel>* or
- *Plane_3<SphericalKernel>*

and depending of these types, the computed *CGAL::Object*s can be assigned to

- *std::pair<Circular_arc_point_3<SphericalKernel>, unsigned>*, where the unsigned integer is the multiplicity of the corresponding intersection point,
- *Circle_3<SphericalKernel>* or
- *Type1*, when *Type1*, *Type2* and *Type3* are equal, and if the three objets *obj1* and *obj2* and *obj3* are equal.

Example

The following example demonstrates the most common use of *intersection* routines with the basic 2D and 3D Kernels.

```
#include <CGAL/intersections.h>

void foo(CGAL::Segment_2<Kernel> seg, CGAL::Line_2<Kernel> line)
{
    CGAL::Object result = CGAL::intersection(seg, line);
    if (const CGAL::Point_2<Kernel> *ipoint = CGAL::object_cast<CGAL::Point_2<Kernel> >(&result)) {


        // handle the point intersection case with *ipoint.


    } else
        if (const CGAL::Segment_2<Kernel> *iseg = CGAL::object_cast<CGAL::Segment_2<Kernel> >(&result)) {


            // handle the segment intersection case with *iseg.


        } else {


            // handle the no intersection case.
        }
}
```

Examples illustrating the use of this function in the case of the 2D Circular Kernel and the 3D Spherical Kernel are presented respectively in Chapters 13 and 14.

See Also

CGAL::do_intersect .... page 2218
CGAL::Object .... page 4167

Function

CGAL::left_turn

bool *left_turn(Point_2<Kernel> p, Point_2<Kernel> q, Point_2<Kernel> r)*

returns *true* iff *p*, *q*, and *r* form a left turn.

See Also

CGAL::collinear . . . . page 700
CGAL::orientation . . . . page 1116
CGAL::right_turn . . . . page 768

CGAL::lexicographically_xyz_smaller

bool *lexicographically_xyz_smaller(Point_3<Kernel> p, Point_3<Kernel> q)*

returns *true* iff *p* is lexicographically smaller than *q* with respect to *xyz* order.

Function

See Also

CGAL::compare_xyz . . . . page 716
CGAL::lexicographically_xyz_smaller_or_equal . . . . page 748

Function

CGAL::lexicographically_xyz_smaller_or_equal

bool *lexicographically_xyz_smaller_or_equal(Point_3<Kernel> p, Point_3<Kernel> q)*

returns *true* iff p is lexicographically not larger than q with respect to *xyz* order.

See Also

CGAL::compare_xyz . . . . page 716
CGAL::lexicographically_xyz_smaller . . . . page 747

CGAL::lexicographically_xy_larger

bool *lexicographically_xy_larger(Point_2<Kernel> p, Point_2<Kernel> q)*

returns *true* iff p is lexicographically larger than q with respect to xy order.

Function

See Also

CGAL::compare_xy . . . . page 714
CGAL::lexicographically_xy_larger_or_equal . . . . page 750
CGAL::lexicographically_xy_smaller . . . . page 751
CGAL::lexicographically_xy_smaller_or_equal . . . . page 752

Function

CGAL::lexicographically_xy_larger_or_equal

bool *lexicographically_xy_larger_or_equal(Point_2<Kernel> p, Point_2<Kernel> q)*

returns *true* iff p is lexicographically not smaller than q with respect to xy order.

See Also

CGAL::compare_xy . . . . page 714
CGAL::lexicographically_xy_larger . . . . page 749
CGAL::lexicographically_xy_smaller . . . . page 751
CGAL::lexicographically_xy_smaller_or_equal . . . . page 752

CGAL::lexicographically_xy_smaller

bool *lexicographically_xy_smaller(Point_2<Kernel> p, Point_2<Kernel> q)*

returns *true* iff *p* is lexicographically smaller than *q* with respect to *xy* order.

See Also

CGAL::compare_xy . . . . page 714
CGAL::lexicographically_xy_larger . . . . page 749
CGAL::lexicographically_xy_larger_or_equal . . . . page 750
CGAL::lexicographically_xy_smaller_or_equal . . . . page 752

Function

Function

CGAL::lexicographically_xy_smaller_or_equal

bool *lexicographically_xy_smaller_or_equal(Point_2<Kernel> p, Point_2<Kernel> q)*

returns *true* iff *p* is lexicographically not larger than *q* with respect to *xy* order.

See Also

CGAL::compare_xy . . . . page 714
CGAL::lexicographically_xy_larger . . . . page 749
CGAL::lexicographically_xy_larger_or_equal . . . . page 750
CGAL::lexicographically_xy_smaller . . . . page 751

CGAL::max_vertex

Point_2<Kernel> *max_vertex(Iso_box_2<Kernel> ir)*

computes the vertex with the lexicographically largest coordinates of the iso rectangle *ir*.

Point_3<Kernel> *max_vertex(Iso_cuboid_3<Kernel> ic)*

computes the vertex with the lexicographically largest coordinates of the iso cuboid *ic*.

CGAL::midpoint

Point_2<Kernel>	*midpoint(Point_2<Kernel> p, Point_2<Kernel> q)*	
		computes the midpoint of the segment *pq*.
Point_3<Kernel>	*midpoint(Point_3<Kernel> p, Point_3<Kernel> q)*	
		computes the midpoint of the segment *pq*.

CGAL::normal

Vector_3<Kernel> *normal(Point_3<Kernel> p, Point_3<Kernel> q, Point_3<Kernel> r)*

computes the normal vector for the vectors *q*-*p* and *r*-*p*.
Precondition: The points *p*, *q*, and *r* must not be collinear.

Function

Function

CGAL::min_vertex

Point_2<Kernel> *min_vertex(Iso_box_2<Kernel> ir)*

computes the vertex with the lexicographically smallest coordinates of the iso rectangle *ir*.

Point_3<Kernel> *min_vertex(Iso_cuboid_3<Kernel> ic)*

computes the vertex with the lexicographically smallest coordinates of the iso cuboid *ic*.

CGAL::operator+

Point_2<Kernel> *operator+(Point_2<Kernel> p, Vector_2<Kernel> v)*

returns the point obtained by translating *p* by vector *v*.

Point_3<Kernel> *operator+(Point_3<Kernel> p, Vector_3<Kernel> v)*

returns a point obtained by translating *p* by vector *v*.

See Also

CGAL::operator- . . . . page 758
*CGAL::operator** . . . . page 759

Function

CGAL::operator-

Vector_2<Kernel>	*operator-(Point_2<Kernel> p, Point_2<Kernel> q)*	
		returns the difference vector between q and p. You can substitute *ORIGIN* for either p or q ,but not for both.
Point_2<Kernel>	*operator-(Point_2<Kernel> p, Vector_2<Kernel> v)*	
		returns the point obtained by translating p by the vector -v.
Vector_3<Kernel>	*operator-(Point_3<Kernel> p, Point_3<Kernel> q)*	
		returns the difference vector between q and p. You can substitute *ORIGIN* for either p or q, but not both.
Point_3<Kernel>	*operator-(Point_3<Kernel> p, Vector_3<Kernel> v)*	
		returns a point obtained by translating p by the vector $-v$.

See Also

CGAL::operator+ . . . . page 757
*CGAL::operator** . . . . page 759

CGAL::operator*

Vector_2<Kernel>	*operator*(Kernel::RT s, Vector_2<Kernel> w)*	
		Multiplication with a scalar from the left.
Vector_3<Kernel>	*operator*(Kernel::RT s, Vector_3<Kernel> w)*	
		Multiplication with a scalar from the left.

See Also

CGAL::operator+ . . . . page 757
CGAL::operator- . . . . page 758

Function

Function

CGAL::opposite

#include <CGAL/enum.h>

Oriented_side	*opposite(Oriented_side o)*	
		returns the opposite side (for example *ON_POSITIVE_SIDE* if *o==ON_NEGATIVE_SIDE*), or *ON_ORIENTED_BOUNDARY* if *o==ON_ORIENTED_BOUNDARY*.
Bounded_side	*opposite(Bounded_side o)*	
		returns the opposite side (for example *BOUNDED_SIDE* if *o==UNBOUNDED_SIDE*), or returns *ON_BOUNDARY* if *o==ON_BOUNDARY*.

CGAL::orthogonal_vector

Vector_3<Kernel> *orthogonal_vector(Plane_3<Kernel> p)*

computes an orthogonal vector of the plane *p*, which is directed to the positive side of this plane.

Vector_3<Kernel> *orthogonal_vector(Point_3<Kernel> p, Point_3<Kernel> q, Point_3<Kernel> r)*

computes an orthogonal vector of the plane defined by *p*, *q* and *r*, which is directed to the positive side of this plane.

Function

Function

CGAL::orientation

Orientation	*orientation(Point_2<Kernel> p, Point_2<Kernel> q, Point_2<Kernel> r)*	
		returns *LEFT_TURN*, if r lies to the left of the oriented line l defined by p and q, returns *RIGHT_TURN* if r lies to the right of l, and returns *COLLINEAR* if r lies on l.
Orientation	*orientation(Vector_2<Kernel> u, Vector_2<Kernel> v)*	
		returns *LEFT_TURN* if u and v form a left turn, returns *RIGHT_TURN* if u and v form a right turn, and returns *COLLINEAR* if u and v are collinear.
Orientation	*orientation(Point_3<Kernel> p, Point_3<Kernel> q, Point_3<Kernel> r, Point_3<Kernel> s)*	
		returns *POSITIVE*, if s lies on the positive side of the oriented plane h defined by p, q, and r, returns *NEGATIVE* if s lies on the negative side of h, and returns *COPLANAR* if s lies on h.
Orientation	*orientation(Vector_3<Kernel> u, Vector_3<Kernel> v, Vector_3<Kernel> w)*	
		returns *POSITIVE* if u, v and w are positively oriented, returns *NEGATIVE* if u, v and w are negatively oriented, and returns *COPLANAR* if u, v and w are coplanar.

See Also

CGAL::collinear . . . . page 700
CGAL::left_turn . . . . page 746
CGAL::right_turn . . . . page 768

CGAL::parallel

Function

bool *parallel(Line_2<Kernel> l1, Line_2<Kernel> l2)*

returns *true*, if *l1* and *l2* are parallel or if one of those (or both) is degenerate.

bool *parallel(Ray_2<Kernel> r1, Ray_2<Kernel> r2)*

returns *true*, if *r1* and *r2* are parallel or if one of those (or both) is degenerate.

bool *parallel(Segment_2<Kernel> s1, Segment_2<Kernel> s2)*

returns *true*, if *s1* and *s2* are parallel or if one of those (or both) is degenerate.

bool *parallel(Line_3<Kernel> l1, Line_3<Kernel> l2)*

returns *true*, if *l1* and *l2* are parallel or if one of those (or both) is degenerate.

bool *parallel(Plane_3<Kernel> h1, Plane_3<Kernel> h2)*

returns *true*, if *h1* and *h2* are parallel or if one of those (or both) is degenerate.

bool *parallel(Ray_3<Kernel> r1, Ray_3<Kernel> r2)*

returns *true*, if *r1* and *r2* are parallel or if one of those (or both) is degenerate.

bool *parallel(Segment_3<Kernel> s1, Segment_3<Kernel> s2)*

returns *true*, if *s1* and *s2* are parallel or if one of those (or both) is degenerate.

CGAL::quotient_cartesian_to_homogeneous

#include <CGAL/cartesian_homogeneous_conversion.h>

Point_2< Homogeneous<RT> >

quotient_cartesian_to_homogeneous(Point_2< Cartesian< Quotient<RT> > > cp)

converts 2d point *cp* with Cartesian representation with number type *Quotient<RT>* into a 2d point with homogeneous representation with number type *RT*.

Point_3< Homogeneous<RT> >

quotient_cartesian_to_homogeneous(Point_3< Cartesian< Quotient<RT> > > cp)

converts 3d point *cp* with Cartesian representation with number type *Quotient<RT>* into a 3d point with homogeneous representation with number type *RT*.

See Also

CGAL::Cartesian<FieldNumberType> .... page 588
CGAL::Cartesian_converter<K1, K2, NTConverter> .... page 589
CGAL::cartesian_to_homogeneous .... page 591
CGAL::Homogeneous<RingNumberType> .... page 1070
CGAL::Homogeneous_converter<K1, K2, RTConverter, FTConverter> .... page 598
CGAL::homogeneous_to_cartesian .... page 599
CGAL::homogeneous_to_quotient_cartesian .... page 600
CGAL::Simple_cartesian<FieldNumberType> .... page 602
CGAL::Simple_homogeneous<RingNumberType> .... page 603

CGAL::radical_plane

Plane_3<Kernel> *radical_plane(Sphere_3<Kernel> s1, Sphere_3<Kernel> s2)*

returns the radical plane of the two spheres.
Precondition: $s1$ and $s2$ are not cocentric.

Function

CGAL::radical_line

Line_2<Kernel> *radical_line(Circle_2<Kernel> c1, Circle_2<Kernel> c2)*

returns the radical line of the two circles.
Precondition: $c1$ and $c2$ are not cocentric.

CGAL::rational_rotation_approximation

#include <CGAL/rational_rotation.h>

template <RingNumberType>
void *rational_rotation_approximation(RingNumberType dirx, RingNumberType diry, RingNumberType* & *sin_num, RingNumberType* & *cos_num, RingNumberType* & *denom, RingNumberType eps_num, RingNumberType eps_den)*

computes integers *sin_num*, *cos_num* and *denom*, such that *sin_num/denom* approximates the sine of direction (*dirx*,*diry*). The difference between the sine and the approximating rational is bounded by *eps_num/eps_den*.
Precondition: $eps\_num \neq 0$.

Implementation

The approximation is based on Farey sequences as described in the rational rotation method presented by Canny and Ressler at the 8th SoCG 1992. We use a slower version which needs no division operation in the approximation.

See Also

CGAL::Aff_transformation_2<Kernel> . . . . page 609

Function

Function

CGAL::right_turn

bool *right_turn(Point_2<Kernel> p, Point_2<Kernel> q, Point_2<Kernel> r)*

returns *true* iff *p*, *q*, and *r* form a right turn.

See Also

CGAL::collinear . . . . page 700
CGAL::left_turn . . . . page 746
CGAL::orientation . . . . page 1116

CGAL::side_of_bounded_circle

Bounded_side *side_of_bounded_circle(Point_2<Kernel> p, Point_2<Kernel> q, Point_2<Kernel> r, Point_2<Kernel> t)*

returns the relative position of point t to the circle defined by p, q and r. The order of the points p, q and r does not matter.
Precondition: p, q and r are not collinear.

Bounded_side *side_of_bounded_circle(Point_2<Kernel> p, Point_2<Kernel> q, Point_2<Kernel> t)*

returns the position of the point t relative to the circle that has pq as its diameter.

See Also

CGAL::coplanar_side_of_bounded_circle .... page 727
CGAL::side_of_oriented_circle .... page 771

Function

Function

CGAL::side_of_bounded_sphere

Bounded_side *side_of_bounded_sphere(Point_3<Kernel> p, Point_3<Kernel> q, Point_3<Kernel> r, Point_3<Kernel> s, Point_3<Kernel> t)*

returns the relative position of point t to the sphere defined by p, q, r, and s. The order of the points p, q, r, and s does not matter.
Precondition: p, q, r and s are not coplanar.

Bounded_side *side_of_bounded_sphere(Point_3<Kernel> p, Point_3<Kernel> q, Point_3<Kernel> r, Point_3<Kernel> t)*

returns the position of the point t relative to the sphere passing through p, q, and r and whose center is in the plane defined by these three points.

Bounded_side *side_of_bounded_sphere(Point_3<Kernel> p, Point_3<Kernel> q, Point_3<Kernel> t)*

returns the position of the point t relative to the sphere that has pq as its diameter.

See Also

CGAL::side_of_oriented_sphere . . . . page 1119

CGAL::side_of_oriented_circle

Oriented_side *side_of_oriented_circle(Point_2<Kernel> p, Point_2<Kernel> q, Point_2<Kernel> r, Point_2<Kernel> test)*

returns the relative position of point *test* to the oriented circle defined by p, q and r. The order of the points p, q and r is important, since it determines the orientation of the implicitly constructed circle.
If p, q and r are collinear, the circle degenerates in a line. *ON_ORIENTED_BOUNDARY* is returned if $test$ is also collinear or if two points are identical, otherwise, *side_of_oriented_circle(r, q, test, p)* is returned.

Function

See Also

CGAL::side_of_bounded_circle .... page 769

CGAL::side_of_oriented_sphere

Oriented_side *side_of_oriented_sphere(Point_3<Kernel> p,*
Point_3<Kernel> q,
Point_3<Kernel> r,
Point_3<Kernel> s,
Point_3<Kernel> test)

returns the relative position of point *test* to the oriented sphere defined by p, q, r and s. The order of the points p, q, r, and s is important, since it determines the orientation of the implicitly constructed sphere. If the points p, q, r and s are positive oriented, positive side is the bounded interior of the sphere.
In case of degeneracies, *ON_ORIENTED_BOUNDARY* is returned if all points are coplanar. Otherwise, there is a cyclic permutation of the five points that puts four non coplanar points first, it is used to answer the predicate: e.g. *side_of_oriented_sphere(q, r, s, test, p)* is returned if q, r, s, and *test* are non coplanar.

See Also

CGAL::side_of_bounded_sphere . . . . page 1118

CGAL::squared_area

Function

Kernel::FT	*squared_area(Point_3<Kernel> p, Point_3<Kernel> q, Point_3<Kernel> r)*	
		returns the squared area of the triangle defined by the points p, q and r.

CGAL::squared_distance

#include <CGAL/squared_distance_2.h>
#include <CGAL/squared_distance_3.h>

Kernel::FT *squared_distance(Type1<Kernel> obj1, Type2<Kernel> obj2)*

computes the square of the Euclidean distance between two geometric objects. For arbitrary geometric objects *obj1* and *obj2* the squared distance is defined as the minimal *squared_distance(p1, p2)*, where *p1* is a point of *obj1* and *p2* is a point of *obj2*. Note that for objects that have an inside (a bounded region), this inside is part of the object. So, the squared distance from a point inside is zero, not the squared distance to the closest point on the boundary.

In 2D, the types *Type1* and *Type2* can be any of the following:

- *Point_2*
- *Line_2*
- *Ray_2*
- *Segment_2*
- *Triangle_2*

In 3D, the types *Type1* and *Type2* can be any of the following:

- *Point_3*
- *Line_3*
- *Ray_3*
- *Segment_3*
- *Plane_3*

See Also

CGAL::compare_distance_to_point . . . . . page 703
CGAL::compare_signed_distance_to_line . . . . . page 705
CGAL::compare_signed_distance_to_plane . . . . . page 706
CGAL::has_larger_distance_to_point . . . . . page 735
CGAL::has_larger_signed_distance_to_line . . . . . page 736
CGAL::has_larger_signed_distance_to_plane . . . . . page 737
CGAL::has_smaller_distance_to_point . . . . . page 738
CGAL::has_smaller_signed_distance_to_line . . . . . page 739
CGAL::has_smaller_signed_distance_to_plane . . . . . page 740

CGAL::squared_radius

FT *squared_radius(Point_2<Kernel> p, Point_2<Kernel> q, Point_2<Kernel> r)*

compute the squared radius of the circle passing through the points p, q, and r.
Precondition: p, q, and r are not collinear.

FT *squared_radius(Point_2<Kernel> p, Point_2<Kernel> q)*

compute the squared radius of the smallest circle passing through p, and q, i.e. one fourth of the squared distance between p and q.

FT *squared_radius(Point_2<Kernel> p)*

compute the squared radius of the smallest circle passing through p, i.e. 0.

FT *squared_radius(Point_3<Kernel> p,*
Point_3<Kernel> q,
Point_3<Kernel> r,
Point_3<Kernel> s)

compute the squared radius of the sphere passing through the points p, q, r and s.
Precondition: p, q, r and s are not coplanar.

FT *squared_radius(Point_3<Kernel> p, Point_3<Kernel> q, Point_3<Kernel> r)*

compute the squared radius of the sphere passing through the points p, q, and r and whose center is in the same plane as those three points.

FT *squared_radius(Point_3<Kernel> p, Point_3<Kernel> q)*

compute the squared radius of the smallest circle passing through p, and q, i.e. one fourth of the squared distance between p and q.

FT *squared_radius(Point_3<Kernel> p)*

compute the squared radius of the smallest circle passing through p, i.e. 0.

See Also

CGAL::Circle_2<Kernel> . . . . page 614
CGAL::Circle_3<Kernel> . . . . page 663
CGAL::Sphere_3<Kernel> . . . . page 666

CGAL::unit_normal

Vector_3<Kernel> *unit_normal(Point_3<Kernel> p, Point_3<Kernel> q, Point_3<Kernel> r)*

computes the unit normal vector for the vectors q-p and r-p.
Precondition: The points p, q, and r must not be collinear.

Function

Function

CGAL::volume

Kernel::FT *volume(Point_3<Kernel> p0,*
Point_3<Kernel> p1,
Point_3<Kernel> p2,
Point_3<Kernel> p3)

Computes the signed volume of the tetrahedron defined by the four points *p0*, *p1*, *p2* and *p3*.

See Also

CGAL::Tetrahedron_3<Kernel> . . . . page 669

CGAL::x_equal

bool *x_equal(Point_2<Kernel> p, Point_2<Kernel> q)*

returns *true*, iff *p* and *q* have the same *x*-coordinate.

bool *x_equal(Point_3<Kernel> p, Point_3<Kernel> q)*

returns *true*, iff *p* and *q* have the same *x*-coordinate.

See Also

CGAL::compare_x . . . . page 712
CGAL::y_equal . . . . page 780
CGAL::z_equal . . . . page 781

Function

CGAL::y_equal

bool *y_equal(Point_2<Kernel> p, Point_2<Kernel> q)*

returns *true*, iff p and q have the same y-coordinate.

bool *y_equal(Point_3<Kernel> p, Point_3<Kernel> q)*

returns *true*, iff p and q have the same y-coordinate.

See Also

CGAL::compare_y . . . . page 719
CGAL::x_equal . . . . page 779
CGAL::z_equal . . . . page 781

CGAL::z_equal

bool *z_equal(Point_3<Kernel> p, Point_3<Kernel> q)*

returns *true*, iff *p* and *q* have the same *z*-coordinate.

See Also

CGAL::compare_z . . . . page 724
CGAL::x_equal . . . . page 779
CGAL::y_equal . . . . page 780

Function

11.14 Kernel Geometric Object Concepts

Kernel::Circle_2

Concept

A type representing circles in two dimensions.

Refines

CopyConstructible, Assignable, DefaultConstructible

See Also

CGAL::Circle_2<Kernel> .... page 614
Kernel::BoundedSide_2 .... page 819
Kernel::ComputeSquaredRadius_2 .... page 872
Kernel::ConstructCenter_2 .... page 896
Kernel::ConstructCircle_2 .... page 900
Kernel::ConstructOppositeCircle_2 .... page 927
Kernel::Equal_2 .... page 996
Kernel::HasOnBoundary_2 .... page 998
Kernel::HasOnBoundedSide_2 .... page 1000
Kernel::HasOnNegativeSide_2 .... page 1002
Kernel::HasOnPositiveSide_2 .... page 1004
Kernel::HasOnUnboundedSide_2 .... page 1006
Kernel::IsDegenerate_2 .... page 1014
Kernel::OrientedSide_2 .... page 1036

Concept

Kernel::Circle_3

A type representing circles in three dimensions.

Refines

CopyConstructible, Assignable, DefaultConstructible

See Also

CGAL::Circle_3<Kernel> .... page 663
Kernel::ComputeApproximateArea_3 .... page 853
Kernel::ComputeApproximateSquaredLength_3 .... page 854
Kernel::ComputeAreaDividedByPi_3 .... page 857
Kernel::ComputeSquaredLengthDividedByPiSquare_3 .... page 871
Kernel::ComputeSquaredRadius_3 .... page 873
Kernel::ConstructBbox_3 .... page 891
Kernel::ConstructCenter_3 .... page 897
Kernel::ConstructCircle_3 .... page 901
Kernel::ConstructSphere_3 .... page 966
Kernel::ConstructPlane_3 .... page 947
Kernel::Equal_3 .... page 997
Kernel::HasOn_3 .... page 1009
Kernel::IsDegenerate_3 .... page 1015

Kernel::Direction_2

A type representing directions in two dimensions.

Refines

CopyConstructible, Assignable, DefaultConstructible

See Also

CGAL::Direction_2<Kernel> . . . page 617
Kernel::CompareAngleWithXAxis_2 . . . page 828
Kernel::ComputeDx_2 . . . page 860
Kernel::ComputeDy_2 . . . page 861
Kernel::ConstructDirection_2 . . . page 908
Kernel::ConstructOppositeDirection_2 . . . page 928
Kernel::ConstructPerpendicularDirection_2 . . . page 942
Kernel::CounterclockwiseInBetween_2 . . . page 987
Kernel::Equal_2 . . . page 996

Concept

Kernel::Direction_3

A type representing directions in three dimensions.

Refines

CopyConstructible, Assignable, DefaultConstructible

See Also

CGAL::Direction_3<Kernel> . . . . page 644
Kernel::ConstructDirection_3 . . . . page 909
Kernel::ConstructOppositeDirection_3 . . . . page 929
Kernel::Equal_2 . . . . page 996

Kernel::IsoCuboid_3

Concept

A type representing isocuboids in three dimensions.

Refines

CopyConstructible, Assignable, DefaultConstructible

See Also

CGAL::Iso_cuboid_3<Kernel> . . . . page 646
Kernel::BoundedSide_3 . . . . page 820
Kernel::ComputeVolume_3 . . . . page 874
Kernel::ConstructIsoCuboid_3 . . . . page 913
Kernel::ConstructVertex_3 . . . . page 981
Kernel::Equal_2 . . . . page 996
Kernel::HasOnBoundary_3 . . . . page 999
Kernel::HasOnBoundedSide_3 . . . . page 1001
Kernel::HasOnUnboundedSide_3 . . . . page 1007
Kernel::IsDegenerate_3 . . . . page 1015

Concept

Kernel::IsoRectangle_2

A type representing iso-rectangles in two dimensions.

Refines

CopyConstructible, Assignable, DefaultConstructible

See Also

CGAL::Iso_rectangle_2<Kernel> . . . page 619
Kernel::ConstructIsoRectangle_2 . . . page 914
Kernel::ComputeXmin_2 . . . page 880
Kernel::ComputeXmax_2 . . . page 882
Kernel::ComputeYmin_2 . . . page 881
Kernel::ComputeYmax_2 . . . page 883
Kernel::BoundedSide_2 . . . page 819
Kernel::ComputeArea_2 . . . page 855
Kernel::ConstructIsoRectangle_2 . . . page 914
Kernel::ConstructVertex_2 . . . page 980
Kernel::DoIntersect_2 . . . page 988
Kernel::Equal_2 . . . page 996
Kernel::HasOnBoundary_2 . . . page 998
Kernel::HasOnBoundedSide_2 . . . page 1000
Kernel::HasOnUnboundedSide_2 . . . page 1006
Kernel::Intersect_2 . . . page 1011
Kernel::IsDegenerate_2 . . . page 1014

Kernel::Line_2

A type representing straight lines (and halfspaces) in two dimensions.

Refines

CopyConstructible, Assignable, DefaultConstructible

See Also

CGAL::Line_2<Kernel> . . . . . . page 622
Kernel::CompareXAtY_2 . . . . . . page 837
Kernel::ComputeSquaredDistance_2 . . . . . . page 867
Kernel::CompareYAtX_2 . . . . . . page 844
Kernel::ConstructBisector_2 . . . . . . page 892
Kernel::ConstructDirection_2 . . . . . . page 908
Kernel::ConstructLine_2 . . . . . . page 916
Kernel::ConstructOppositeLine_2 . . . . . . page 930
Kernel::ConstructPerpendicularLine_2 . . . . . . page 943
Kernel::ConstructPointOn_2 . . . . . . page 949
Kernel::ConstructProjectedPoint_2 . . . . . . page 953
Kernel::DoIntersect_2 . . . . . . page 988
Kernel::Equal_2 . . . . . . page 996
Kernel::HasOnNegativeSide_2 . . . . . . page 1002
Kernel::HasOnPositiveSide_2 . . . . . . page 1004
Kernel::HasOn_2 . . . . . . page 1008
Kernel::Intersect_2 . . . . . . page 1011
Kernel::IsDegenerate_2 . . . . . . page 1014
Kernel::IsHorizontal_2 . . . . . . page 1017
Kernel::IsVertical_2 . . . . . . page 1018
Kernel::OrientedSide_2 . . . . . . page 1036

Concept

Kernel::Line_3

A type representing straight lines in three dimensions.

Refines

CopyConstructible, Assignable, DefaultConstructible

See Also

Kernel::Object_2

A type representing different types of objects in two dimensions.

Refines

CopyConstructible, Assignable, DefaultConstructible

See Also

CGAL::Object . . . . page 4167
Kernel::Assign_2 . . . . page 817
Kernel::ConstructObject_2 . . . . page 925
Kernel::Intersect_2 . . . . page 1011

Concept

Concept

Kernel::Object_3

A type representing different types of objects in three dimensions.

Refines

CopyConstructible, Assignable, DefaultConstructible

See Also

CGAL::Object . . . . page 4167
Kernel::Assign_3 . . . . page 818
Kernel::ConstructObject_3 . . . . page 926
Kernel::Intersect_3 . . . . page 1012

Kernel::Plane_3

Concept

A type representing planes (and half-spaces) in three dimensions.

Refines

CopyConstructible, Assignable, DefaultConstructible

See Also

CGAL::Plane_3<Kernel> . . . page 651
Kernel::ComputeSquaredDistance_3 . . . page 868
Kernel::ConstructBaseVector_3 . . . page 889
Kernel::ConstructBisector_3 . . . page 893
Kernel::ConstructLiftedPoint_3 . . . page 915
Kernel::ConstructOppositePlane_3 . . . page 932
Kernel::ConstructOrthogonalVector_3 . . . page 941
Kernel::ConstructPerpendicularLine_3 . . . page 944
Kernel::ConstructPerpendicularPlane_3 . . . page 945
Kernel::ConstructPlane_3 . . . page 947
Kernel::ConstructPointOn_3 . . . page 950
Kernel::ConstructProjectedPoint_3 . . . page 954
Kernel::ConstructProjectedXYPoint_2 . . . page 955
Kernel::DoIntersect_3 . . . page 989
Kernel::Equal_3 . . . page 997
Kernel::HasOnNegativeSide_3 . . . page 1003
Kernel::HasOnPositiveSide_3 . . . page 1005
Kernel::HasOn_3 . . . page 1009
Kernel::Intersect_3 . . . page 1012
Kernel::IsDegenerate_3 . . . page 1015
Kernel::LessSignedDistanceToPlane_3 . . . page 1024
Kernel::OrientedSide_3 . . . page 1037

Concept

Kernel::Point_2

A type representing points in two dimensions.

Refines

CopyConstructible, Assignable, DefaultConstructible

See Also

Kernel::Angle_2 . . . page 809
Kernel::AreOrderedAlongLine_2 . . . page 811
Kernel::AreStrictlyOrderedAlongLine_2 . . . page 815
Kernel::Collinear_2 . . . page 826
Kernel::CollinearAreOrderedAlongLine_2 . . . page 821
Kernel::CollinearAreStrictlyOrderedAlongLine_2 . . . page 823
Kernel::CompareDistance_2 . . . page 831
Kernel::CompareXAtY_2 . . . page 837
Kernel::CompareXY_2 . . . page 840
Kernel::CompareX_2 . . . page 842
Kernel::CompareYAtX_2 . . . page 844
Kernel::CompareY_2 . . . page 846
Kernel::CompareYX_2 . . . page 848
Kernel::ComputeSquaredDistance_2 . . . page 867
Kernel::ComputeSquaredRadius_2 . . . page 872
Kernel::ComputeX_2 . . . page 875
Kernel::ComputeY_2 . . . page 877
Kernel::ComputeHx_2 . . . page 862
Kernel::ComputeHy_2 . . . page 863
Kernel::ConstructBisector_2 . . . page 892
Kernel::ConstructCircumcenter_2 . . . page 903
Kernel::ConstructLiftedPoint_3 . . . page 915
Kernel::ConstructMidpoint_2 . . . page 920
Kernel::ConstructPointOn_2 . . . page 949
Kernel::ConstructPoint_2 . . . page 951
Kernel::ConstructProjectedPoint_2 . . . page 953
Kernel::ConstructProjectedXYPoint_2 . . . page 955
Kernel::ConstructTranslatedPoint_2 . . . page 973
Kernel::DoIntersect_2 . . . page 988
Kernel::Equal_2 . . . page 996
Kernel::EqualX_2 . . . page 991
Kernel::EqualY_2 . . . page 993
Kernel::LeftTurn_2 . . . page 1019
Kernel::LessDistanceToPoint_2 . . . page 1020
Kernel::LessRotateCCW_2 . . . page 1022
Kernel::LessSignedDistanceToLine_2 . . . page 1023
Kernel::LessX_2 . . . page 1028
Kernel::LessXY_2 . . . page 1026
Kernel::LessY_2 . . . page 1031
Kernel::LessYX_2 . . . page 1030
Kernel::Orientation_2 . . . page 1034

Concept

Kernel::Point_3

A type representing points in three dimensions.

Refines

CopyConstructible, Assignable, DefaultConstructible

See Also

Kernel::Angle_3 . . . page 810
Kernel::AreOrderedAlongLine_3 . . . page 812
Kernel::AreStrictlyOrderedAlongLine_3 . . . page 816
Kernel::Collinear_3 . . . page 827
Kernel::CollinearAreOrderedAlongLine_3 . . . page 822
Kernel::CollinearAreStrictlyOrderedAlongLine_3 . . . page 824
Kernel::CompareDistance_3 . . . page 832
Kernel::CompareXYZ_3 . . . page 839
Kernel::CompareXY_3 . . . page 841
Kernel::CompareX_3 . . . page 843
Kernel::CompareY_3 . . . page 847
Kernel::CompareZ_3 . . . page 849
Kernel::ComputeSquaredDistance_3 . . . page 868
Kernel::ComputeSquaredRadius_3 . . . page 873
Kernel::ComputeX_3 . . . page 876
Kernel::ComputeY_3 . . . page 878
Kernel::ComputeZ_3 . . . page 879
Kernel::ConstructBisector_3 . . . page 893
Kernel::ConstructCentroid_3 . . . page 899
Kernel::ConstructCircumcenter_3 . . . page 904
Kernel::ConstructLiftedPoint_3 . . . page 915
Kernel::ConstructMidpoint_3 . . . page 921
Kernel::ConstructPointOn_3 . . . page 950
Kernel::ConstructPoint_3 . . . page 952
Kernel::ConstructProjectedPoint_3 . . . page 954
Kernel::ConstructTranslatedPoint_3 . . . page 974
Kernel::CoplanarOrientation_3 . . . page 984
Kernel::CoplanarSideOfBoundedCircle_3 . . . page 985
Kernel::Coplanar_3 . . . page 986
Kernel::EqualXY_3 . . . page 990
Kernel::EqualX_3 . . . page 992
Kernel::EqualY_3 . . . page 994
Kernel::EqualZ_3 . . . page 995
Kernel::Equal_2 . . . page 996
Kernel::LessDistanceToPoint_3 . . . page 1021
Kernel::LessSignedDistanceToPlane_3 . . . page 1024
Kernel::LessXYZ_3 . . . page 1025
Kernel::LessXY_3 . . . page 1027
Kernel::LessX_3 . . . page 1029
Kernel::LessY_3 . . . page 1032
Kernel::LessZ_3 . . . page 1033

Kernel::Ray_2

A type representing rays in two dimensions.

Refines

CopyConstructible, Assignable, DefaultConstructible

See Also

CGAL::Ray_2<Kernel> . . . page 629
Kernel::CollinearHasOn_2 . . . page 825
Kernel::ComputeSquaredDistance_2 . . . page 867
Kernel::ConstructDirection_2 . . . page 908
Kernel::ConstructLine_2 . . . page 916
Kernel::ConstructOppositeRay_2 . . . page 933
Kernel::ConstructPointOn_2 . . . page 949
Kernel::ConstructRay_2 . . . page 958
Kernel::ConstructSource_2 . . . page 965
Kernel::ConstructSecondPoint_2 . . . page 962
Kernel::DoIntersect_2 . . . page 988
Kernel::Equal_2 . . . page 996
Kernel::HasOn_2 . . . page 1008
Kernel::Intersect_2 . . . page 1011
Kernel::IsDegenerate_2 . . . page 1014
Kernel::IsHorizontal_2 . . . page 1017
Kernel::IsVertical_2 . . . page 1018

Kernel::Ray_3

A type representing rays in three dimensions.

Refines

CopyConstructible, Assignable, DefaultConstructible

See Also

CGAL::Ray_3<Kernel> . . . . page 659
Kernel::ComputeSquaredDistance_3 . . . . page 868
Kernel::ConstructDirection_3 . . . . page 909
Kernel::ConstructLine_3 . . . . page 917
Kernel::ConstructOppositeRay_3 . . . . page 934
Kernel::ConstructPlane_3 . . . . page 947
Kernel::ConstructPointOn_3 . . . . page 950
Kernel::ConstructRay_3 . . . . page 959
Kernel::DoIntersect_3 . . . . page 989
Kernel::Equal_3 . . . . page 997
Kernel::HasOn_3 . . . . page 1009
Kernel::Intersect_3 . . . . page 1012
Kernel::IsDegenerate_3 . . . . page 1015

Concept

Concept

Kernel::Segment_2

A type representing segments in two dimensions.

Refines

CopyConstructible, Assignable, DefaultConstructible

See Also

CGAL::Segment_2<Kernel> . . . . page 631
Kernel::CollinearHasOn_2 . . . . page 825
Kernel::ComputeSquaredDistance_2 . . . . page 867
Kernel::ComputeSquaredLength_2 . . . . page 869
Kernel::ConstructDirection_2 . . . . page 908
Kernel::ConstructLine_2 . . . . page 916
Kernel::ConstructOppositeSegment_2 . . . . page 935
Kernel::ConstructPointOn_2 . . . . page 949
Kernel::ConstructSegment_2 . . . . page 963
Kernel::ConstructSource_2 . . . . page 965
Kernel::ConstructTarget_2 . . . . page 971
Kernel::ConstructVertex_2 . . . . page 980
Kernel::DoIntersect_2 . . . . page 988
Kernel::Equal_2 . . . . page 996
Kernel::HasOn_2 . . . . page 1008
Kernel::Intersect_2 . . . . page 1011
Kernel::IsDegenerate_2 . . . . page 1014
Kernel::IsHorizontal_2 . . . . page 1017
Kernel::IsVertical_2 . . . . page 1018

Kernel::Segment_3

A type representing segments in three dimensions.

Refines

CopyConstructible, Assignable, DefaultConstructible

See Also

CGAL::Segment_3<Kernel> . . . . . page 661
Kernel::ComputeSquaredDistance_3 . . . . . page 868
Kernel::ComputeSquaredLength_3 . . . . . page 870
Kernel::ConstructDirection_3 . . . . . page 909
Kernel::ConstructLine_3 . . . . . page 917
Kernel::ConstructOppositeSegment_3 . . . . . page 936
Kernel::ConstructPlane_3 . . . . . page 947
Kernel::ConstructPointOn_3 . . . . . page 950
Kernel::ConstructSegment_3 . . . . . page 964
Kernel::ConstructVertex_3 . . . . . page 981
Kernel::DoIntersect_3 . . . . . page 989
Kernel::Equal_3 . . . . . page 997
Kernel::HasOn_3 . . . . . page 1009
Kernel::Intersect_3 . . . . . page 1012
Kernel::IsDegenerate_3 . . . . . page 1015

Concept

Concept

Kernel::Sphere_3

A type representing spheres in three dimensions.

Refines

CopyConstructible, Assignable, DefaultConstructible

See Also

CGAL::Sphere_3<Kernel> .... page 666
Kernel::BoundedSide_3 .... page 820
Kernel::ComputeSquaredRadius_3 .... page 873
Kernel::ConstructCenter_3 .... page 897
Kernel::ConstructOppositeSphere_3 .... page 937
Kernel::ConstructRadicalPlane_3 .... page 957
Kernel::ConstructSphere_3 .... page 966
Kernel::Equal_2 .... page 996
Kernel::HasOnBoundary_3 .... page 999
Kernel::HasOnBoundedSide_3 .... page 1001
Kernel::HasOnNegativeSide_3 .... page 1003
Kernel::HasOnPositiveSide_3 .... page 1005
Kernel::HasOnUnboundedSide_3 .... page 1007
Kernel::IsDegenerate_3 .... page 1015
Kernel::OrientedSide_3 .... page 1037

Kernel::Tetrahedron_3

A type representing tetrahedra in three dimensions.

Refines

CopyConstructible, Assignable, DefaultConstructible

See Also

CGAL::Tetrahedron_3<Kernel> . . . . . . page 669
Kernel::BoundedSide_3 . . . . . . page 820
Kernel::ComputeVolume_3 . . . . . . page 874
Kernel::ConstructCentroid_3 . . . . . . page 899
Kernel::ConstructTetrahedron_3 . . . . . . page 972
Kernel::ConstructVertex_3 . . . . . . page 981
Kernel::Equal_2 . . . . . . page 996
Kernel::HasOnBoundary_3 . . . . . . page 999
Kernel::HasOnBoundedSide_3 . . . . . . page 1001
Kernel::HasOnNegativeSide_3 . . . . . . page 1003
Kernel::HasOnPositiveSide_3 . . . . . . page 1005
Kernel::HasOnUnboundedSide_3 . . . . . . page 1007
Kernel::IsDegenerate_3 . . . . . . page 1015
Kernel::OrientedSide_3 . . . . . . page 1037

Concept

Concept

Kernel::Triangle_2

A type representing triangles in two dimensions.

Refines

CopyConstructible, Assignable, DefaultConstructible

See Also

CGAL::Triangle_2<Kernel> . . . . . page 633
Kernel::BoundedSide_2 . . . . . page 819
Kernel::ComputeArea_2 . . . . . page 855
Kernel::ComputeSquaredDistance_2 . . . . . page 867
Kernel::ConstructCentroid_2 . . . . . page 898
Kernel::ConstructOppositeTriangle_2 . . . . . page 938
Kernel::ConstructTriangle_2 . . . . . page 975
Kernel::ConstructVertex_2 . . . . . page 980
Kernel::DoIntersect_2 . . . . . page 988
Kernel::Equal_2 . . . . . page 996
Kernel::HasOnBoundary_2 . . . . . page 998
Kernel::HasOnBoundedSide_2 . . . . . page 1000
Kernel::HasOnNegativeSide_2 . . . . . page 1002
Kernel::HasOnPositiveSide_2 . . . . . page 1004
Kernel::HasOnUnboundedSide_2 . . . . . page 1006
Kernel::Intersect_2 . . . . . page 1011
Kernel::IsDegenerate_2 . . . . . page 1014
Kernel::OrientedSide_2 . . . . . page 1036

Kernel::Triangle_3

A type representing triangles in three dimensions.

Refines

CopyConstructible, Assignable, DefaultConstructible

See Also

CGAL::Triangle_3<Kernel> . . . . page 671
Kernel::ComputeSquaredArea_3 . . . . page 866
Kernel::ConstructCentroid_3 . . . . page 899
Kernel::ConstructSupportingPlane_3 . . . . page 970
Kernel::ConstructTriangle_3 . . . . page 976
Kernel::ConstructVertex_3 . . . . page 981
Kernel::DoIntersect_3 . . . . page 989
Kernel::Equal_3 . . . . page 997
Kernel::HasOn_3 . . . . page 1009
Kernel::IsDegenerate_3 . . . . page 1015

Concept

Kernel::Vector_2

Definition

A type representing vectors in two dimensions.

Refines

CopyConstructible, Assignable, DefaultConstructible

See Also

CGAL::Vector_2<Kernel> . . . . page 635
Kernel::ComputeDeterminant_2 . . . . page 858
Kernel::ComputeX_2 . . . . page 875
Kernel::ComputeY_2 . . . . page 877
Kernel::ComputeHx_2 . . . . page 862
Kernel::ComputeHy_2 . . . . page 863
Kernel::ConstructDirection_2 . . . . page 908
Kernel::ConstructOppositeVector_2 . . . . page 939
Kernel::ConstructPerpendicularVector_2 . . . . page 946
Kernel::ConstructScaledVector_2 . . . . page 960
Kernel::ConstructDividedVector_2 . . . . page 910
Kernel::ConstructSumOfVectors_2 . . . . page 968
Kernel::ConstructDifferenceOfVectors_2 . . . . page 906
Kernel::ConstructVector_2 . . . . page 978
Kernel::Equal_2 . . . . page 996
Kernel::Orientation_2 . . . . page 1034

Kernel::Vector_3

Concept

Definition

A type representing vectors in three dimensions.

Refines

CopyConstructible, Assignable, DefaultConstructible

See Also

CGAL::Vector_3<Kernel> . . . . . . page 673
Kernel::ComputeDeterminant_3 . . . . . . page 859
Kernel::ComputeX_3 . . . . . . page 876
Kernel::ComputeY_3 . . . . . . page 878
Kernel::ComputeZ_3 . . . . . . page 879
Kernel::ConstructCrossProductVector_3 . . . . . . page 905
Kernel::ConstructDirection_3 . . . . . . page 909
Kernel::ConstructOppositeVector_3 . . . . . . page 940
Kernel::ConstructOrthogonalVector_3 . . . . . . page 941
Kernel::ConstructScaledVector_3 . . . . . . page 961
Kernel::ConstructDividedVector_3 . . . . . . page 911
Kernel::ConstructSumOfVectors_3 . . . . . . page 969
Kernel::ConstructDifferenceOfVectors_3 . . . . . . page 907
Kernel::ConstructVector_3 . . . . . . page 979
Kernel::Equal_3 . . . . . . page 997
Kernel::Orientation_3 . . . . . . page 1035

11.15 Kernel Function Object Concepts

Kernel::Angle_2

A model for this must provide:

Angle *fo.operator()(Kernel::Vector_2 u, Kernel::Vector_2 v)*

returns *OBTUSE*, *RIGHT* or *ACUTE* depending on the angle formed by the two vectors *u* and *v*.

Angle *fo.operator()(Kernel::Point_2 p, Kernel::Point_2 q, Kernel::Point_2 r)*

returns *OBTUSE*, *RIGHT* or *ACUTE* depending on the angle formed by the three points *p*, *q*, *r* (*q* being the vertex of the angle). The returned value is the same as *operator()(p - q, r - q)*.

Angle *fo.operator()(Kernel::Point_2 p, Kernel::Point_2 q, Kernel::Point_2 r, Kernel::Point_2 s)*

returns *OBTUSE*, *RIGHT* or *ACUTE* depending on the angle formed by the two vectors *pq*, *rs*. The returned value is the same as *operator()(q - p, s - r)*.

See Also

CGAL::angle . . . . page 689

Functor
Concept

Kernel::Angle_3

A model for this must provide:

Angle	*fo.operator()(Kernel::Point_3 p, Kernel::Point_3 q, Kernel::Point_3 r)*
	returns *OBTUSE*, *RIGHT* or *ACUTE* depending on the angle formed by the three points p, q, r (q being the vertex of the angle).

Refines

AdaptableFunctor (with three arguments)

See Also

CGAL::angle . . . . page 689

Kernel::AreOrderedAlongLine_2

A model for this must provide:

bool *fo.operator()(Kernel::Point_2 p, Kernel::Point_2 q, Kernel::Point_2 r)*

returns *true*, iff the three points are collinear and q lies between p and r. Note that *true* is returned, if $q{=}{=}p$ or $q{=}{=}r$.

Functor
Concept

Refines

AdaptableFunctor (with three arguments)

See Also

CGAL::are_ordered_along_line .... page 690

Kernel::AreOrderedAlongLine_3

A model for this must provide:

bool	*fo.operator()(Kernel::Point_3 p, Kernel::Point_3 q, Kernel::Point_3 r)*
	returns *true*, iff the three points are collinear and q lies between p and r. Note that *true* is returned, if $q{=}{=}p$ or $q{=}{=}r$.

Refines

AdaptableFunctor (with three arguments)

See Also

CGAL::are_ordered_along_line . . . . page 690

Kernel::AreParallel_2

A model for this must provide:

bool *fo.operator()(Kernel::Line_2 l1, Kernel::Line_2 l2)*

returns *true*, if *l1* and *l2* are parallel or if one of those (or both) is degenerate.

bool *fo.operator()(Kernel::Ray_2 r1, Kernel::Ray_2 r2)*

returns *true*, if *r1* and *r2* are parallel or if one of those (or both) is degenerate.

bool *fo.operator()(Kernel::Segment_2 s1, Kernel::Segment_2 s2)*

returns *true*, if *s1* and *s2* are parallel or if one of those (or both) is degenerate.

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::parallel . . . . page 763

Kernel::AreParallel_3

A model for this must provide:

bool *fo.operator()(Kernel::Line_3 l1, Kernel::Line_3 l2)*

returns *true*, if *l1* and *l2* are parallel or if one of those (or both) is degenerate.

bool *fo.operator()(Kernel::Plane_3 h1, Kernel::Plane_3 h2)*

returns *true*, if *h1* and *h2* are parallel or if one of those (or both) is degenerate.

bool *fo.operator()(Kernel::Ray_3 r1, Kernel::Ray_3 r2)*

returns *true*, if *r1* and *r2* are parallel or if one of those (or both) is degenerate.

bool *fo.operator()(Kernel::Segment_3 s1, Kernel::Segment_3 s2)*

returns *true*, if *s1* and *s2* are parallel or if one of those (or both) is degenerate.

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::parallel . . . . page 763

Kernel::AreStrictlyOrderedAlongLine_2

A model for this must provide:

bool *fo.operator()(Kernel::Point_2 p, Kernel::Point_2 q, Kernel::Point_2 r)*

returns *true*, iff the three points are collinear and q lies strictly between p and r. Note that *false* is returned, if q==p or q==r.

Functor
Concept

Refines

AdaptableFunctor (with three arguments)

See Also

CGAL::are_strictly_ordered_along_line . . . . page 691

Functor
Concept

Kernel::AreStrictlyOrderedAlongLine_3

A model for this must provide:

bool	*fo.operator()(Kernel::Point_3 p, Kernel::Point_3 q, Kernel::Point_3 r)*
	returns *true*, iff the three points are collinear and q lies strictly between p and r. Note that *false* is returned, if $q{=}{=}p$ or $q{=}{=}r$.

Refines

AdaptableFunctor (with three arguments)

See Also

CGAL::are_strictly_ordered_along_line . . . . page 691

Kernel::Assign_2

A model for this must provide:

template <class T>
bool *fo.operator()(T& t, Kernel::Object_2 o)*

assigns o to t if o was constructed from an object of type T. Returns *true*, if the assignment was possible.

Functor
Concept

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::Object . . . . page 4167
Kernel::Object_2 . . . . page 791
Kernel::Intersect_2 . . . . page 1011

Kernel::Assign_3

A model for this must provide:

template <class T>
bool *fo.operator()(T& t, Kernel::Object_3 o)*

assigns o to t if o was constructed from an object of type T. Returns *true*, if the assignment was possible.

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::Object . . . . page 4167
Kernel::Object_3 . . . . page 792
Kernel::Intersect_3 . . . . page 1012

Kernel::BoundedSide_2

A model for this must provide:

Bounded_side *fo.operator()(Kernel::Circle_2 c, Kernel::Point_2 p)*

returns either *ON_UNBOUNDED_SIDE*, *ON_BOUNDED_SIDE*, or the constant *ON_BOUNDARY*, depending on where point p is relative to circle c.

Bounded_side *fo.operator()(Kernel::Triangle_2 t, Kernel::Point_2 p)*

returns either *ON_UNBOUNDED_SIDE*, *ON_BOUNDED_SIDE*, or the constant *ON_BOUNDARY*, depending on where point p is relative to triangle t.

Bounded_side *fo.operator()(Kernel::Iso_rectangle_2 r, Kernel::Point_2 p)*

returns either *ON_UNBOUNDED_SIDE*, *ON_BOUNDED_SIDE*, or the constant *ON_BOUNDARY*, depending on where point p is relative to rectangle r.

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::Circle_2<Kernel> . . . . page 614
CGAL::Triangle_2<Kernel> . . . . page 633
CGAL::Iso_rectangle_2<Kernel> . . . . page 619

Functor
Concept

Kernel::BoundedSide_3

A model for this must provide:

Bounded_side *fo.operator()(Kernel::Sphere_3 s, Kernel::Point_3 p)*

returns either *ON_UNBOUNDED_SIDE*, *ON_BOUNDED_SIDE*, or the constant *ON_BOUNDARY*, depending on where point *p* is with respect to sphere *s*.

Bounded_side *fo.operator()(Kernel::Tetrahedron_3 t, Kernel::Point_3 p)*

returns either *ON_UNBOUNDED_SIDE*, *ON_BOUNDED_SIDE*, or the constant *ON_BOUNDARY*, depending on where point *p* is with respect to tetrahedron *t*.

Bounded_side *fo.operator()(Kernel::Iso_cuboid_3 c, Kernel::Point_3 p)*

returns either *ON_UNBOUNDED_SIDE*, *ON_BOUNDED_SIDE*, or the constant *ON_BOUNDARY*, depending on where point *p* is with respect to iso-cuboid *c*.

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::Iso_cuboid_3<Kernel> . . . . page 646
CGAL::Sphere_3<Kernel> . . . . page 666
CGAL::Tetrahedron_3<Kernel> . . . . page 669

Kernel::CollinearAreOrderedAlongLine_2

A model for this must provide:

bool *fo.operator()(Kernel::Point_2 p, Kernel::Point_2 q, Kernel::Point_2 r)*

returns *true*, iff q lies between p and r.
Precondition: p, q and r are collinear.

Functor
Concept

Refines

AdaptableFunctor (with three arguments)

See Also

CGAL::collinear_are_ordered_along_line . . . . page 701

Functor
Concept

Kernel::CollinearAreOrderedAlongLine_3

A model for this must provide:

bool	*fo.operator()(Kernel::Point_3 p, Kernel::Point_3 q, Kernel::Point_3 r)*
	returns *true*, iff q lies between p and r. *Precondition*: p, q and r are collinear.

Refines

AdaptableFunctor (with three arguments)

See Also

CGAL::collinear_are_ordered_along_line . . . . page 701

Kernel::CollinearAreStrictlyOrderedAlongLine_2

A model for this must provide:

bool	*fo.operator()(Kernel::Point_2 p, Kernel::Point_2 q, Kernel::Point_2 r)*
	returns *true*, iff *q* lies strictly between *p* and *r*. *Precondition*: *p, q* and *r* are collinear.

Refines

AdaptableFunctor (with three arguments)

See Also

CGAL::collinear_are_strictly_ordered_along_line . . . . page 702

Kernel::CollinearAreStrictlyOrderedAlongLine_3

A model for this must provide:

bool *fo.operator()(Kernel::Point_3 p, Kernel::Point_3 q, Kernel::Point_3 r)*

returns *true*, iff q lies strictly between p and r.
Precondition: p, q and r are collinear.

Refines

AdaptableFunctor (with three arguments)

See Also

CGAL::collinear_are_strictly_ordered_along_line . . . . page 702

Kernel::CollinearHasOn_2

A model for this must provide:

bool *fo.operator()(Kernel::Ray_2 r, Kernel::Point_2 p)*

checks if point p is on r.
Precondition: p is on the supporting line of r.

bool *fo.operator()(Kernel::Segment_2 s, Kernel::Point_2 p)*

checks if point p is on s.
Precondition: p is on the supporting line of s.

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::Ray_2<Kernel> . . . . page 629
CGAL::Segment_2<Kernel> . . . . page 631

Kernel::Collinear_2

A model for this must provide:

bool	*fo.operator()(Kernel::Point_2 p, Kernel::Point_2 q, Kernel::Point_2 r)*
	returns *true*, if *p*, *q*, and *r* are collinear.

Refines

AdaptableFunctor (with three arguments)

See Also

CGAL::collinear . . . . page 700

Kernel::Collinear_3

A model for this must provide:

bool	*fo.operator()(Kernel::Point_3 p, Kernel::Point_3 q, Kernel::Point_3 r)*	
		returns *true*, if p, q, and r are collinear.

Refines

AdaptableFunctor (with three arguments)

See Also

CGAL::collinear . . . . page 700

Kernel::CompareAngleWithXAxis_2

A model for this must provide:

Comparison_result *fo.operator()(Kernel::Direction_2 d, Kernel::Direction_2 e)*

compares the angles between the positive x-axis and the directions in counterclockwise order.

Refines

AdaptableFunctor (with two arguments)

Kernel::CompareDihedralAngle_3

A model for this must provide:

Comparison_result *fo.operator()(K::Point_3 a1, K::Point_3 b1, K::Point_3 c1, K::Point_3 d1, K::FT cosine)*

compares the dihedral angles θ_1 and θ_2, where θ_1 is the dihedral angle, in $[0,\pi]$, of the tetrahedron (a_1,b_1,c_1,d_1) at the edge (a_1,b_1), and θ_2 is the angle in $[0,\pi]$ such that $cos(\theta_2) = cosine$. The result is the same as *operator()(b1-a1, c1-a1, d1-a1, cosine)*.
Precondition: a_1, b_1, c_1 are not collinear, and a_1, b_1, d_1 are not collinear.

Comparison_result *fo.operator()(K::Point_3 a1, K::Point_3 b1, K::Point_3 c1, K::Point_3 d1, K::Point_3 a2, K::Point_3 b2, K::Point_3 c2, K::Point_3 d2)*

compares the dihedral angles θ_1 and θ_2, where θ_i is the dihedral angle in the tetrahedron (a_i,b_i,c_i,d_i) at the edge (a_i,b_i). These two angles are computed in $[0,\pi]$. The result is the same as *operator()(b1-a1, c1-a1, d1-a1, b2-a2, c2-a2, d2-a2)*.
Precondition: For $i \in \{1,2\}$, a_i, b_i, c_i are not collinear, and a_i, b_i, d_i are not collinear.

Comparison_result *fo.operator()(K::Vector_3 u1, K::Vector_3 v1, K::Vector_3 w1, K::FT cosine)*

compares the dihedral angles θ_1 and θ_2, where θ_1 is the dihedral angle, in $[0,\pi]$, between the vectorial planes defined by (u_1,v_1) and (u_1,w_1), and θ_2 is the angle in $[0,\pi]$ such that $cos(\theta_2) = cosine$.
Precondition: u_1 and v_1 are not collinear, and u_1 and w_1 are not collinear.

Comparison_result *fo.operator()(K::Vector_3 u1, K::Vector_3 v1, K::Vector_3 w1, K::Vector_3 u2, K::Vector_3 v2,* *K::Vector_3 w2)*

compares the dihedral angles θ_1 and θ_2, where θ_i is the dihedral angle between the vectorial planes defined by (u_i, v_i) and (u_i, w_i). These two angles are computed in $[0, \pi]$.
Precondition: For $i \in \{1, 2\}$, u_i and v_i are not collinear, and u_i and w_i are not collinear.

Kernel::CompareDistance_2

A model for this must provide:

Comparison_result *fo.operator()(Type1 obj1, Type2 obj2, Type3 obj3)*

compares the squared distance between *obj1* and *obj2* to the squared distance between *obj1* and *obj3*

Comparison_result *fo.operator()(Type1 obj1, Type2 obj2, Type3 obj3, Type4 obj4)*

compares the squared distance between *obj1* and *obj2* to the squared distance between *obj3* and *obj4*

for all tuples of types *Type1*, *Type2*, *Type3* and *Type4* in the following set of types:

- *Kernel::Point_2*
- *Kernel::Line_2*
- *Kernel::Ray_2*
- *Kernel::Segment_2*
- *Kernel::Triangle_2*

Refines

AdaptableFunctor (with three arguments)

See Also

Kernel::CompareSquaredDistance_2 . . . . page 834
CGAL::compare_distance_to_point . . . . page 703
CGAL::compare_squared_distance . . . . page 710

Functor
Concept

Kernel::CompareDistance_3

A model for this must provide:

Comparison_result *fo.operator()(Type1 obj1, Type2 obj2, Type3 obj3)*

compares the squared distance between *obj1* and *obj2* to the squared distance between *obj1* and *obj3*

Comparison_result *fo.operator()(Type1 obj1, Type2 obj2, Type3 obj3, Type4 obj4)*

compares the squared distance between *obj1* and *obj2* to the squared distance between *obj3* and *obj4*

for all tuples of types *Type1*, *Type2*, *Type3* and *Type4* in the following set of types:

- *Kernel::Point_3*
- *Kernel::Line_3*
- *Kernel::Ray_3*
- *Kernel::Segment_3*
- *Kernel::Plane_3*

Refines

AdaptableFunctor (with three arguments)

See Also

Kernel::CompareSquaredDistance_3 . . . . page 835
CGAL::compare_distance_to_point . . . . page 703
CGAL::compare_squared_distance . . . . page 710

Kernel::CompareSlope_2

A model for this must provide:

Comparison_result *fo.operator()(Kernel::Line_2 l1, Kernel::Line_2 l2)*

compares the slopes of the lines *l1* and *l2*

Comparison_result *fo.operator()(Kernel::Segment_2 s1, Kernel::Segment_2 s2)*

compares the slopes of the segments *s1* and *s2*

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::compare_slopes . . . . page 707

Functor
Concept

Kernel::CompareSquaredDistance_2

A model for this must provide:

Comparison_result *fo.operator()(Type1 obj1, Type2 obj2, Kernel::FT d2)*

compares the squared distance between the two geometrical objects *obj1* and *obj2* to the value *d2*

for all pairs *Type1* and *Type2*, where the types *Type1* and *Type2* can be any of the following:

- *Kernel::Point_2*
- *Kernel::Line_2*
- *Kernel::Ray_2*
- *Kernel::Segment_2*
- *Kernel::Triangle_2*

Refines

AdaptableFunctor (with three arguments)

See Also

CGAL::compare_distance_to_point . . . . page 703
CGAL::compare_squared_distance . . . . page 710

Kernel::CompareSquaredDistance_3

A model for this must provide:

Comparison_result *fo.operator()(Type1 obj1, Type2 obj2, Kernel::FT d2)*

compares the squared distance between the two geometrical objects *obj1* and *obj2* to the value *d2*

for all pairs *Type1* and *Type2*, where the types *Type1* and *Type2* can be any of the following:

- *Kernel::Point_3*
- *Kernel::Line_3*
- *Kernel::Ray_3*
- *Kernel::Segment_3*
- *Kernel::Plane_3*

Refines

AdaptableFunctor (with three arguments)

See Also

Kernel::CompareDistance_3 . . . . page 832
CGAL::compare_distance_to_point . . . . page 703
CGAL::compare_squared_distance . . . . page 710

Kernel::CompareSquaredRadius_3

A model for this must provide:

Comparison_result *fo.operator()(Kernel::Point_3 p, Kernel::FT sr)*

compares the squared radius of the sphere of radius 0 centered at *p* to *sr*. This returns the opposite sign of *sr*.

Comparison_result *fo.operator()(Kernel::Point_3 p, Kernel::Point_3 q, Kernel::FT sr)*

compares the squared radius of the sphere defined by the points *p* and *q* to *sr*.

Comparison_result *fo.operator()(Kernel::Point_3 p, Kernel::Point_3 q, Kernel::Point_3 r, Kernel::FT sr)*

compares the squared radius of the sphere defined by the points *p*, *q*, and *r* to *sr*.

Comparison_result *fo.operator()(Kernel::Point_3 p, Kernel::Point_3 q, Kernel::Point_3 r, Kernel::Point_3 s, Kernel::FT sr)*

compares the squared radius of the sphere defined by the points *p*, *q*, , *r*, and *s* to *sr*.

Refines

AdaptableFunctor (with three arguments)

See Also

CGAL::compare_squared_radius . . . . page 711

Kernel::CompareXAtY_2

A model for this must provide:

Comparison_result *fo.operator()(Kernel::Point_2 p, Kernel::Line_2 h)*

compares the x-coordinates of p and the horizontal projection of p on h (Figure 11.5 (a)).

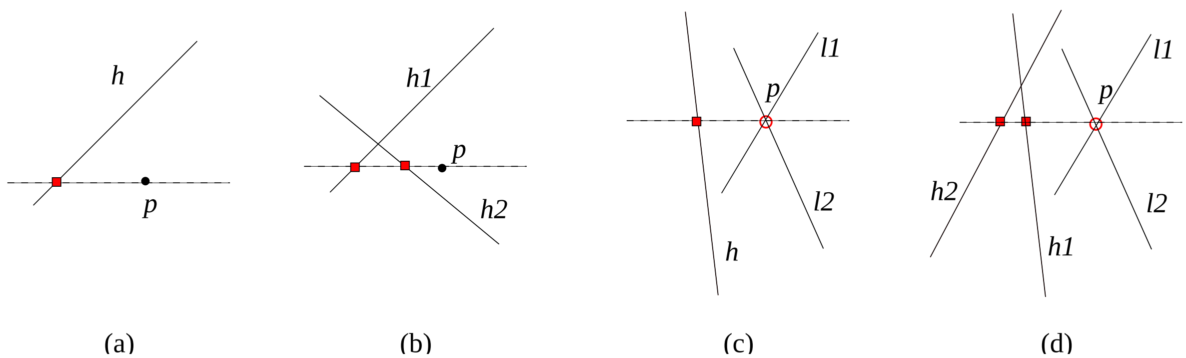


Figure 11.5: Comparison of the x-coordinates of the (implicitly given) points in the boxes, at a y-coordinate. The y-coordinate is either given explicitly (disc) or implicitly (circle).

Comparison_result *fo.operator()(Kernel::Point_2 p, Kernel::Line_2 h1, Kernel::Line_2 h2)*

compares the x-coordinates of the horizontal projection of p on $h1$ and on $h2$ (Figure 11.5 (b)).

Comparison_result *fo.operator()(Kernel::Line_2 l1, Kernel::Line_2 l2, Kernel::Line_2 h)*

Let p be the intersection of lines $l1$ and $l2$. This function compares the x-coordinates of p and the horizontal projection of p on h (Figure 11.5 (c)).

Comparison_result *fo.operator()(Kernel::Line_2 l1,*
Kernel::Line_2 l2,
Kernel::Line_2 h1,
Kernel::Line_2 h2)

Let p be the intersection of lines $l1$ and $l2$. This function compares the x-coordinates of the horizontal projection of p on $h1$ and on $h2$ (Figure 11.5 (d)).

Refines

AdaptableFunctor (with three arguments)

See Also

CGAL::compare_x_at_y . . . . page 717

Kernel::CompareXYZ_3

A model for this must provide:

Comparison_result *fo.operator()(Kernel::Point_3 p, Kernel::Point_3 q)*

Compares the Cartesian coordinates of points p and q lexicographically in xyz order: first x-coordinates are compared, if they are equal, y-coordinates are compared. If they are equal, z-coordinates are compared.

Functor
Concept

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::compare_xyz . . . . page 716

Kernel::CompareXY_2

A model for this must provide:

Comparison_result *fo.operator()(Kernel::Point_2 p, Kernel::Point_2 q)*

Compares the Cartesian coordinates of points p and q lexicographically in xy order: first x-coordinates are compared, if they are equal, y-coordinates are compared.

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::compare_xy . . . . page 714
Kernel::CompareYX_2 . . . . page 848

Kernel::CompareXY_3

A model for this must provide:

Comparison_result	*fo.operator()(Kernel::Point_3 p, Kernel::Point_3 q)*	
		Compares the Cartesian coordinates of points p and q lexicographically in xy order: first x-coordinates are compared, if they are equal, y-coordinates are compared.

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::compare_xy . . . . page 714

Kernel::CompareX_2

A model for this must provide:

Comparison_result *fo.operator()(Kernel::Point_2 p, Kernel::Point_2 q)*

compares the Cartesian *x*-coordinates of points *p* and *q*

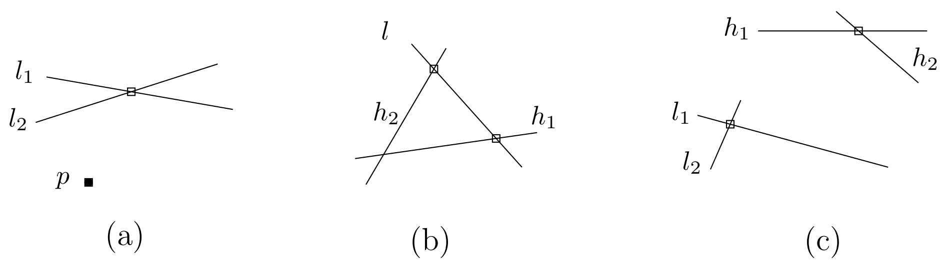


Figure 11.6: Comparison of the *x* or *y*-coordinates of the (implicitly given) points in the boxes.

Comparison_result *fo.operator()(Kernel::Point_2 p, Kernel::Line_2 l1, Kernel::Line_2 l2)*

compares the *x*-coordinates of *p* and the intersection of lines *l*1 and *l*2 (Figure 11.6 (a)).

Comparison_result *fo.operator()(Kernel::Line_2 l, Kernel::Line_2 h1, Kernel::Line_2 h2)*

compares the *x*-coordinates of the intersection of line *l* with line *h*1 and with line *h*2 (Figure 11.6 (b)).

Comparison_result *fo.operator()(Kernel::Line_2 l1, Kernel::Line_2 l2, Kernel::Line_2 h1, Kernel::Line_2 h2)*

compares the *x*-coordinates of the intersection of lines *l*1 and *l*2 and the intersection of lines *h*1 and *h*2 (Figure 11.6 (c)).

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::compare_x . . . . page 712

Kernel::CompareX_3

A model for this must provide:

Comparison_result *fo.operator()(Kernel::Point_3 p, Kernel::Point_3 q)*

Compares the Cartesian x-coordinates of points p and q

Functor
Concept

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::compare_x . . . . page 712

Functor
Concept

Kernel::CompareYAtX_2

A model for this must provide:

Comparison_result *fo.operator()(Kernel::Point_2 p, Kernel::Line_2 h)*

compares the y-coordinates of p and the vertical projection of p on h (Figure 11.7 (d)).
Precondition: h is not vertical.

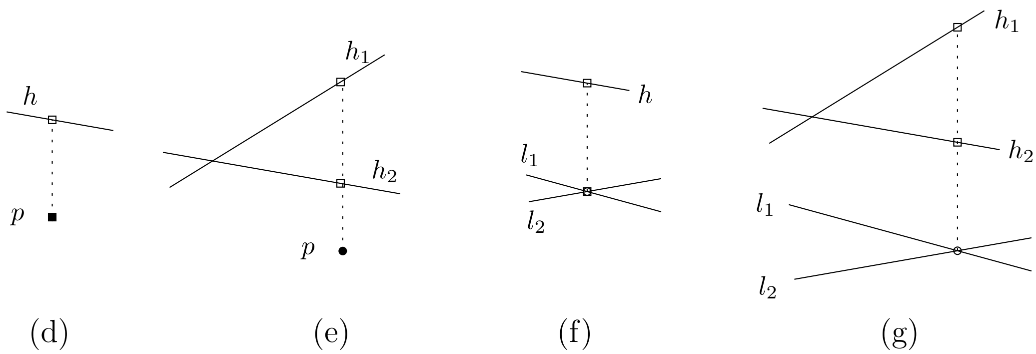


Figure 11.7: Comparison of the y-coordinates of the (implicitly given) points in the boxes, at an x-coordinate. The x-coordinate is either given explicitly (disc) or implicitly (circle).

Comparison_result *fo.operator()(Kernel::Point_2 p, Kernel::Line_2 h1, Kernel::Line_2 h2)*

This function compares the y-coordinates of the vertical projection of p on *h1* and on *h2* (Figure 11.4 (e)).
Precondition: *h1* and *h2* are not vertical.

Comparison_result *fo.operator()(Kernel::Line_2 l1, Kernel::Line_2 l2, Kernel::Line_2 h)*

Let p be the intersection of lines $l1$ and $l2$. This function compares the y-coordinates of p and the vertical projection of p on h (Figure 11.4 (f)).
Precondition: *l1*, *l2* intersect and h is not vertical.

Comparison_result *fo.operator()(Kernel::Line_2 l1,*
Kernel::Line_2 l2,
Kernel::Line_2 h1,
Kernel::Line_2 h2)

Let p be the intersection of lines $l1$ and $l2$. This function compares the y-coordinates of the vertical projection of p on *h1* and on *h2* (Figure 11.4 (g)).
Precondition: *l1* and *l2* intersect; *h1* and *h2* are not vertical.

Comparison_result *fo.operator()(Kernel::Point_2 p, Kernel::Segment_2 s)*

compares the *y*-coordinates of *p* and the vertical projection of *p* on *s*. If *s* is vertical, then return *EQUAL* when *p* lies on *s*, *SMALLER* when *p* lies under s, and *LARGER* otherwise.
Precondition: *p* is within the x range of *s*.

Comparison_result *fo.operator()(Kernel::Point_2 p, Kernel::Segment_2 s1, Kernel::Segment_2 s2)*

This function compares the *y*-coordinates of the vertical projection of *p* on *s1* and on *s2*. If *s1* or *s2* is vertical, then return *EQUAL* if they intersect, otherwise return *SMALLER* if *s1* lies below *s2*, and return *LARGER* otherwise.
Precondition: *p* is within the x range of *s1* and *s2*.

Refines

AdaptableFunctor (with three arguments)

See Also

CGAL::compare_y_at_x . . . . page 721

Kernel::CompareY_2

A model for this must provide:

Comparison_result *fo.operator()(Kernel::Point_2 p, Kernel::Point_2 q)*

Compares the Cartesian y-coordinates of points p and q

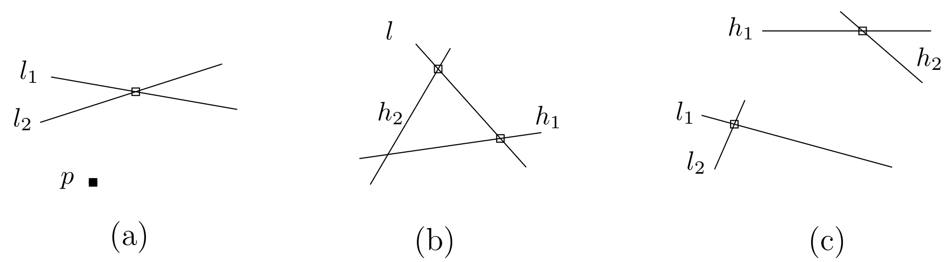


Figure 11.8: Comparison of the x or y-coordinates of the (implicitly given) points in the boxes.

Comparison_result *fo.operator()(Kernel::Point_2 p, Kernel::Line_2 l1, Kernel::Line_2 l2)*

compares the y-coordinates of p and the intersection of lines $l1$ and $l2$ (Figure 11.8 (a)).

Comparison_result *fo.operator()(Kernel::Line_2 l, Kernel::Line_2 h1, Kernel::Line_2 h2)*

compares the y-coordinates of the intersection of line l with line $h1$ and with line $h2$ (Figure 11.8 (b)).

Comparison_result *fo.operator()(Kernel::Line_2 l1, Kernel::Line_2 l2, Kernel::Line_2 h1, Kernel::Line_2 h2)*

compares the y-coordinates of the intersection of lines $l1$ and $l2$ and the intersection of lines $h1$ and $h2$ (Figure 11.8 (c)).

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::compare_y . . . . page 719

Kernel::CompareY_3

A model for this must provide:

Comparison_result	*fo.operator()(Kernel::Point_3 p, Kernel::Point_3 q)*
	Compares the Cartesian y-coordinates of points p and q

Functor
Concept

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::compare_y . . . . page 719

Kernel::CompareYX_2

A model for this must provide:

Comparison_result *fo.operator()(Kernel::Point_2 p, Kernel::Point_2 q)*

Compares the Cartesian coordinates of points p and q lexicographically in yx order: first y-coordinates are compared, if they are equal, x-coordinates are compared.

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::compare_yx . . . . page 723
Kernel::CompareXY_2 . . . . page 840

Kernel::CompareZ_3

A model for this must provide:

Comparison_result *fo.operator()(Kernel::Point_3 p, Kernel::Point_3 q)*

Compares the Cartesian z-coordinates of points p and q

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::compare_z . . . . page 724

Kernel::ComputeA_2

A model for this must provide:

Kernel::FT	*fo.operator()(Kernel::Line_2 l) const*	
		returns the coefficient a of the line with equation $ax + by + c = 0$.

Refines

AdaptableFunctor

Kernel::ComputeB_2

A model for this must provide:

Kernel::FT	*fo.operator()(Kernel::Line_2 l) const*	
		returns the coefficient b of the line with equation $ax + by + c = 0$.

Refines

AdaptableFunctor

Functor
Concept

Kernel::ComputeC_2

A model for this must provide:

Kernel::FT *fo.operator()(Kernel::Line_2 l) const*

returns the coefficient c of the line with equation $ax + by + c = 0$.

Refines

AdaptableFunctor

Kernel::ComputeApproximateArea_3

A model for this must provide:

double	*fo.operator()(Kernel::Circle_3 c)*	
		returns an approximation of the area of *c*.

Functor
Concept

Refines

AdaptableFunctor (with one argument)

See Also

CGAL::Circle_3<Kernel> . . . . page 663

Functor
Concept

Kernel::ComputeApproximateSquaredLength_3

A model for this must provide:

double	*fo.operator()(Kernel::Circle_3 c)*	
		returns an approximation of the squared length (i.e. perimeter) of *c*.

Refines

AdaptableFunctor (with one argument)

See Also

CGAL::Circle_3<Kernel> . . . . page 663

Kernel::ComputeArea_2

A model for this must provide:

Kernel::FT *fo.operator()(Kernel::Point_2 p, Kernel::Point_2 q, Kernel::Point_2 r)*

returns the signed area of the triangle defined by the points p, q and r.

Kernel::FT *fo.operator()(Kernel::Iso_rectangle_2 r)*

returns the area of r.

Kernel::FT *fo.operator()(Kernel::Triangle_2 t)*

returns the signed area of t.

Refines

AdaptableFunctor (with one argument)

See Also

CGAL::Iso_rectangle_2<Kernel> . . . . page 619
CGAL::Triangle_2<Kernel> . . . . page 633

Functor Concept

Kernel::ComputeArea_3

A model for this must provide:

Kernel::FT *fo.operator()(Kernel::Triangle_3 t)*

returns the area of t. This requires that *Kernel::FT* supports the *sqrt* operation.

Kernel::FT *fo.operator()(Kernel::Point_3 p, Kernel::Point_3 q, Kernel::Point_3 r)*

returns the area of the triangle p, q, r. This requires that *Kernel::FT* supports the *sqrt* operation.

Refines

AdaptableFunctor (with one argument)

See Also

CGAL::Triangle_3<Kernel> . . . . page 671

Kernel::ComputeAreaDividedByPi_3

A model for this must provide:

Kernel::FT *fo.operator()(Kernel::Circle_3 c)*

returns the area of c, divided by π.

Refines

AdaptableFunctor (with one argument)

See Also

CGAL::Circle_3<Kernel> . . . . page 663

Functor Concept

Functor
Concept

Kernel::ComputeDeterminant_2

A model for this must provide:

Kernel::FT	*fo.operator()(Kernel::Vector_2 v, Kernel::Vector_2 w)*
	returns the determinant of the two vectors *v* and *w*.

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::Vector_2<Kernel> . . . . page 635
CGAL::determinant . . . . page 729

Kernel::ComputeDeterminant_3

A model for this must provide:

Kernel::FT *fo.operator()(Kernel::Vector_3 u, Kernel::Vector_3 v, Kernel::Vector_3 w)*

returns the determinant of the three vectors *u*, *v* and *w*.

Refines

AdaptableFunctor (with three arguments)

See Also

CGAL::Vector_3<Kernel> . . . . page 673
CGAL::determinant . . . . page 729

Kernel::ComputeDx_2

A model for this must provide:

Kernel::FT *fo.operator()(Kernel::Direction_2 v) const*

returns an x-coordinate of the direction.

Refines

AdaptableFunctor

Kernel::ComputeDy_2

A model for this must provide:

Kernel::FT	*fo.operator()(Kernel::Direction_2 v) const*
	returns an y-coordinate of the direction.

Refines

AdaptableFunctor

Kernel::ComputeHx_2

A model for this must provide:

Kernel::FT	*fo.operator()(Kernel::Point_2 p) const*	
		returns the homogeneous x-coordinate of the point.
Kernel::FT	*fo.operator()(Kernel::Vector_2 v) const*	
		returns the homogeneous x-coordinate of the vector.

Refines

AdaptableFunctor

Kernel::ComputeHy_2

A model for this must provide:

Kernel::FT *fo.operator()(Kernel::Point_2 p) const*

returns the homogeneous y-coordinate of the point.

Kernel::FT *fo.operator()(Kernel::Vector_2 v) const*

returns the homogeneous y-coordinate of the vector.

Refines

AdaptableFunctor

Functor
Concept

Kernel::ComputeScalarProduct_2

A model for this must provide:

Kernel::FT	*fo.operator()(Kernel::Vector_2 v, Kernel::Vector_2 w)*	
		returns the scalar (inner) product of the two vectors v and w.

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::Vector_2<Kernel> . . . . page 635

Kernel::ComputeScalarProduct_3

A model for this must provide:

Kernel::FT	*fo.operator()(Kernel::Vector_3 v, Kernel::Vector_3 w)*	
		returns the scalar (inner) product of the two vectors v and w.

Functor
Concept

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::Vector_3<Kernel> . . . . page 673

Kernel::ComputeSquaredArea_3

A model for this must provide:

Kernel::FT *fo.operator()(Kernel::Triangle_3 t)*

returns the square of the area of t.

Kernel::FT *fo.operator()(Kernel::Point_3 p, Kernel::Point_3 q, Kernel::Point_3 r)*

returns the square of the area of the triangle p, q, r.

Refines

AdaptableFunctor (with one argument)

See Also

CGAL::Triangle_3<Kernel> . . . . page 671

Kernel::ComputeSquaredDistance_2

A model for this must provide:

Kernel::FT *fo.operator()(Type1 obj1, Type2 obj2)*

returns the squared distance between two geometrical objects of type *Type1* and *Type2*

for all pairs *Type1* and *Type2*, where the types *Type1* and *Type2* can be any of the following:

- *Kernel::Point_2*
- *Kernel::Line_2*
- *Kernel::Ray_2*
- *Kernel::Segment_2*
- *Kernel::Triangle_2*

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::squared_distance . . . . page 1120

Functor
Concept

Kernel::ComputeSquaredDistance_3

A model for this must provide:

Kernel::FT	*fo.operator()(Type1 obj1, Type2 obj2)*	
		returns the squared distance between two geometrical objects of type *Type1* and *Type2*

for all pairs *Type1* and *Type2*, where the types *Type1* and *Type2* can be any of the following:

- *Kernel::Point_3*
- *Kernel::Line_3*
- *Kernel::Ray_3*
- *Kernel::Segment_3*
- *Kernel::Plane_3*

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::squared_distance . . . . page 1120

Kernel::ComputeSquaredLength_2

A model for this must provide:

Kernel::FT	*fo.operator()(Kernel::Vector_2 v)*	
		returns the squared length of v.
Kernel::FT	*fo.operator()(Kernel::Segment_2 s)*	
		returns the squared length of s.

Refines

AdaptableFunctor (with one argument)

See Also

CGAL::Vector_2<Kernel> . . . . page 635
CGAL::Segment_2<Kernel> . . . . page 631

Functor
Concept

Kernel::ComputeSquaredLength_3

A model for this must provide:

Kernel::FT	*fo.operator()(Kernel::Vector_3 v)*	
		returns the squared length of *v*.
Kernel::FT	*fo.operator()(Kernel::Segment_3 s)*	
		returns the squared length of *s*.

Refines

AdaptableFunctor (with one argument)

See Also

CGAL::Vector_3<Kernel> .... page 673
CGAL::Segment_3<Kernel> .... page 661

Kernel::ComputeSquaredLengthDividedByPiSquare_3

A model for this must provide:

Kernel::FT	*fo.operator()(Kernel::Circle_3 c)*	
		returns the squared length of c, divided by π^2.

Refines

AdaptableFunctor (with one argument)

See Also

CGAL::Circle_3<Kernel> . . . . page 663

Kernel::ComputeSquaredRadius_2

A model for this must provide:

Kernel::FT *fo.operator()(Kernel::Circle_2 c)*

returns the squared radius of c.

Kernel::FT *fo.operator()(Kernel::Point_2 p, Kernel::Point_2 q, Kernel::Point_2 r)*

returns the squared radius of the circle passing through p, q and r.
Precondition: p, q and r are not collinear.

Kernel::FT *fo.operator()(Kernel::Point_2 p, Kernel::Point_2 q)*

returns the squared radius of the smallest circle passing through p, and q, i.e. one fourth of the squared distance between p and q.

Kernel::FT *fo.operator()(Kernel::Point_2 p)*

returns the squared radius of the smallest circle passing through p, i.e. 0.

Refines

AdaptableFunctor (with one argument)

See Also

CGAL::Circle_2<Kernel> . . . . page 614
CGAL::squared_radius . . . . page 775

Kernel::ComputeSquaredRadius_3

A model for this must provide:

Kernel::FT	*fo.operator()(Kernel::Sphere_3 s)*	
		returns the squared radius of s.
Kernel::FT	*fo.operator()(Kernel::Circle_3 c)*	
		returns the squared radius of c.
Kernel::FT	*fo.operator()(Kernel::Point_3 p, Kernel::Point_3 q, Kernel::Point_3 r, Kernel::Point_3 s)*	
		returns the squared radius of the sphere passing through p, q, r and s. *Precondition*: p, q, r and s are not coplanar.
Kernel::FT	*fo.operator()(Kernel::Point_3 p, Kernel::Point_3 q, Kernel::Point_3 r)*	
		returns the squared radius of the sphere passing through p, q and r, and whose center is in the plane defined by these three points.
Kernel::FT	*fo.operator()(Kernel::Point_3 p, Kernel::Point_3 q)*	
		returns the squared radius of the smallest circle passing through p, and q, i.e. one fourth of the squared distance between p and q.
Kernel::FT	*fo.operator()(Kernel::Point_3 p)*	
		returns the squared radius of the smallest circle passing through p, i.e. 0.

Refines

AdaptableFunctor (with one argument)

See Also

CGAL::Sphere_3<Kernel> . . . . page 666
CGAL::Circle_3<Kernel> . . . . page 663
CGAL::squared_radius . . . . page 775

Kernel::ComputeVolume_3

A model for this must provide:

Kernel::FT *fo.operator()(Kernel::Iso_cuboid_3 c)*

returns the volume of *c*.

Kernel::FT *fo.operator()(Kernel::Tetrahedron_3 t)*

returns the signed volume of *t*.

Kernel::FT *fo.operator()(Kernel::Point_3 p0,*
Kernel::Point_3 p1,
Kernel::Point_3 p2,
Kernel::Point_3 p3)

returns the signed volume of the tetrahedron defined by the four points *p0*, *p1*, *p2*, *p3*.

Refines

AdaptableFunctor (with one argument)

See Also

CGAL::Iso_cuboid_3<Kernel> . . . . page 646
CGAL::Tetrahedron_3<Kernel> . . . . page 669

Kernel::ComputeX_2

A model for this must provide:

Kernel::FT *fo.operator()(Kernel::Point_2 p) const*

returns the x-coordinate of the point.

Kernel::FT *fo.operator()(Kernel::Vector_2 v) const*

returns the x-coordinate of the vector.

Refines

AdaptableFunctor

Kernel::ComputeX_3

A model for this must provide:

Kernel::FT *fo.operator()(Kernel::Point_3 p) const*

returns the x-coordinate of the point.

Kernel::FT *fo.operator()(Kernel::Vector_3 v) const*

returns the x-coordinate of the vector.

Refines

AdaptableFunctor

Kernel::ComputeY_2

A model for this must provide:

Kernel::FT *fo.operator()(Kernel::Point_2 p) const*

returns the y-coordinate of the point.

Kernel::FT *fo.operator()(Kernel::Vector_2 v) const*

returns the y-coordinate of the vector.

Refines

AdaptableFunctor

Kernel::ComputeY_3

A model for this must provide:

Kernel::FT *fo.operator()(Kernel::Point_3 p) const*

returns the y-coordinate of the point.

Kernel::FT *fo.operator()(Kernel::Vector_3 v) const*

returns the y-coordinate of the vector.

Refines

AdaptableFunctor

Kernel::ComputeZ_3

A model for this must provide:

Kernel::FT	*fo.operator()(Kernel::Point_3 p) const*	
		returns the z-coordinate of the point.
Kernel::FT	*fo.operator()(Kernel::Vector_3 v) const*	
		returns the z-coordinate of the vector.

Refines

AdaptableFunctor

Functor
Concept

Kernel::ComputeXmin_2

A model for this must provide:

Kernel::FT	*fo.operator()(Kernel::Iso_rectangle_2 r) const*	
		returns the smallest *x*-coordinate of the iso-rectangle.

Refines

AdaptableFunctor

Kernel::ComputeYmin_2

A model for this must provide:

Kernel::FT *fo.operator()(Kernel::Iso_rectangle_2 r) const*

returns the smallest *y*-coordinate of the iso-rectangle.

Refines

AdaptableFunctor

Kernel::ComputeXmax_2

A model for this must provide:

Kernel::FT	*fo.operator()(Kernel::Iso_rectangle_2 r) const*	
		returns the largest x-coordinate of the iso-rectangle.

Refines

AdaptableFunctor

Kernel::ComputeYmax_2

A model for this must provide:

Kernel::FT	*fo.operator()(Kernel::Iso_rectangle_2 r) const*
	returns the largest y-coordinate of the iso-rectangle.

Refines

AdaptableFunctor

Kernel::ComputeYAtX_2

A model for this must provide:

Kernel::FT	*fo.operator()(Kernel::Line_2 l, Kernel::FT x) const*	

returns the y-coordinate of the point at l with given x-coordinate.
Precondition: l is not vertical.

Refines

AdaptableFunctor

See Also

CGAL::compare_y_at_x . . . . page 721

Kernel::ConstructBarycenter_2

A model for this must provide:

Kernel::Point_2 *fo.operator()(Kernel::Point_2 p1, Kernel::FT w1, Kernel::Point_2 p2)*

compute the barycenter of the points $p1$ and $p2$ with corresponding weights $w1$ and $1 - w1$.

Kernel::Point_2 *fo.operator()(Kernel::Point_2 p1,
Kernel::FT w1,
Kernel::Point_2 p2,
Kernel::FT w2)*

compute the barycenter of the points $p1$ and $p2$ with corresponding weights $w1$ and $w2$.
Precondition: w1+w2 != 0.

Kernel::Point_2 *fo.operator()(Kernel::Point_2 p1,
Kernel::FT w1,
Kernel::Point_2 p2,
Kernel::FT w2,
Kernel::Point_2 p3)*

compute the barycenter of the points $p1$, $p2$ and $p3$ with corresponding weights $w1$, $w2$ and $1 - w1 - w2$.

Kernel::Point_2 *fo.operator()(Kernel::Point_2 p1,
Kernel::FT w1,
Kernel::Point_2 p2,
Kernel::FT w2,
Kernel::Point_2 p3,
Kernel::FT w3)*

compute the barycenter of the points $p1$, $p2$ and $p3$ with corresponding weights $w1$, $w2$ and $w3$.
Precondition: w1+w2+w3 != 0.

Kernel::Point_2 *fo.operator()(Kernel::Point_2 p1,
Kernel::FT w1,
Kernel::Point_2 p2,
Kernel::FT w2,
Kernel::Point_2 p3,
Kernel::FT w3,
Kernel::Point_2 p4)*

compute the barycenter of the points $p1$, $p2$, $p3$ and $p4$ with corresponding weights $w1$, $w2$, $w3$ and $1 - w1 - w2 - w3$.

Kernel::Point_2	*fo.operator()(Kernel::Point_2 p1,* *Kernel::FT w1,* *Kernel::Point_2 p2,* *Kernel::FT w2,* *Kernel::Point_2 p3,* *Kernel::FT w3,* *Kernel::Point_2 p4,* *Kernel::FT w4)*	

compute the barycenter of the points $p1$, $p2$, $p3$ and $p4$ with corresponding weights $w1$, $w2$, $w3$ and $w4$.
Precondition: w1+w2+w3+w4 != 0.

Refines

AdaptableFunctor (with three arguments)

See Also

Kernel::ConstructCentroid_2 . . . . page 898
CGAL::centroid . . . . page 3969
CGAL::barycenter . . . . page 3971

Kernel::ConstructBarycenter_3

A model for this must provide:

Kernel::Point_3 *fo.operator()(Kernel::Point_3 p1, Kernel::FT w1, Kernel::Point_3 p2)*

compute the barycenter of the points $p1$ and $p2$ with corresponding weights $w1$ and $1-w1$.

Kernel::Point_3 *fo.operator()(Kernel::Point_3 p1, Kernel::FT w1, Kernel::Point_3 p2, Kernel::FT w2)*

compute the barycenter of the points $p1$ and $p2$ with corresponding weights $w1$ and $w2$.
Precondition: w1+w2 != 0.

Kernel::Point_3 *fo.operator()(Kernel::Point_3 p1, Kernel::FT w1, Kernel::Point_3 p2, Kernel::FT w2, Kernel::Point_3 p3)*

compute the barycenter of the points $p1$, $p2$ and $p3$ with corresponding weights $w1$, $w2$ and $1-w1-w2$.

Kernel::Point_3 *fo.operator()(Kernel::Point_3 p1, Kernel::FT w1, Kernel::Point_3 p2, Kernel::FT w2, Kernel::Point_3 p3, Kernel::FT w3)*

compute the barycenter of the points $p1$, $p2$ and $p3$ with corresponding weights $w1$, $w2$ and $w3$.
Precondition: w1+w2+w3 != 0.

Kernel::Point_3 *fo.operator()(Kernel::Point_3 p1, Kernel::FT w1, Kernel::Point_3 p2, Kernel::FT w2, Kernel::Point_3 p3, Kernel::FT w3, Kernel::Point_3 p4)*

compute the barycenter of the points $p1$, $p2$, $p3$ and $p4$ with corresponding weights $w1$, $w2$, $w3$ and $1-w1-w2-w3$.

Kernel::Point_3	*fo.operator()(Kernel::Point_3 p1, Kernel::FT w1, Kernel::Point_3 p2, Kernel::FT w2, Kernel::Point_3 p3, Kernel::FT w3, Kernel::Point_3 p4, Kernel::FT w4)*	

compute the barycenter of the points $p1$, $p2$, $p3$ and $p4$ with corresponding weights $w1$, $w2$, $w3$ and $w4$.
Precondition: w1+w2+w3+w4 != 0.

Refines

AdaptableFunctor (with three arguments)

See Also

Kernel::ConstructCentroid_3 . . . . page 899
CGAL::centroid . . . . page 3969
CGAL::barycenter . . . . page 3971

Kernel::ConstructBaseVector_3

A model for this must provide:

Kernel::Vector_3	*fo.operator()(Kernel::Plane_3 h, int index)*

when *index* == 1, returns a vector *b1* that is orthogonal to the normal *n* to plane *h*; when *index* == 2, returns a vector *b2* that is orthogonal to *n* and *b1* and such that for an arbitrary point *p* on the plane *h*, the orientation of *p*, *p* + *b1*, *p* + *b2*, and *p* + *n* is positive.

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::Plane_3<Kernel> . . . . page 651

Functor
Concept

Kernel::ConstructBbox_2

A model for this must provide:

CGAL::Bbox_2 *fo.operator()(Kernel::Point_2 p)*

returns a bounding box of *p*.

CGAL::Bbox_2 *fo.operator()(Kernel::Segment_2 s)*

returns a bounding box of *s*.

CGAL::Bbox_2 *fo.operator()(Kernel::Triangle_2 t)*

returns a bounding box of *t*.

CGAL::Bbox_2 *fo.operator()(Kernel::Iso_rectangle_2 i)*

returns a bounding box of *i*.

CGAL::Bbox_2 *fo.operator()(Kernel::Circle_2 c)*

returns a bounding box of *c*.

Refines

AdaptableFunctor (with one argument)

Kernel::ConstructBbox_3

A model for this must provide:

CGAL::Bbox_3 *fo.operator()(Kernel::Circle_3 c)*

returns a bounding box of *c*.

CGAL::Bbox_3 *fo.operator()(Kernel::Point_3 p)*

returns a bounding box of *p*.

CGAL::Bbox_3 *fo.operator()(Kernel::Segment_3 s)*

returns a bounding box of *s*.

CGAL::Bbox_3 *fo.operator()(Kernel::Triangle_3 t)*

returns a bounding box of *t*.

CGAL::Bbox_3 *fo.operator()(Kernel::Tetrahedron_3 t)*

returns a bounding box of *t*.

CGAL::Bbox_3 *fo.operator()(Kernel::Iso_Cuboid_3 i)*

returns a bounding box of *i*.

CGAL::Bbox_3 *fo.operator()(Kernel::Sphere_3 s)*

returns a bounding box of *s*.

Refines

AdaptableFunctor (with one argument)

Kernel::ConstructBisector_2

A model for this must provide:

Kernel::Line_2 *fo.operator()(Kernel::Point_2 p, Kernel::Point_2 q)*

constructs the bisector of p and q. The bisector is oriented in such a way that p lies on its positive side.
Precondition: p and q are not equal.

Kernel::Line_2 *fo.operator()(Kernel::Line_2 l1, Kernel::Line_2 l2)*

constructs the bisector of the two lines $l1$ and $l2$. In the general case, the bisector has the direction of the vector which is the sum of the normalized directions of the two lines, and which passes through the intersection of *l1* and *l2*. If *l1* and *l2* are parallel, then the bisector is defined as the line which has the same direction as *l1*, and which is at the same distance from *l1* and *l2*. This function requires that *Kernel::RT* supports the *sqrt()* operation.

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::bisector . . . . page 696

Kernel::ConstructBisector_3

A model for this must provide:

Kernel::Plane_3 *fo.operator()(Kernel::Point_3 p, Kernel::Point_3 q)*

constructs the bisector plane of p and q. The bisector is oriented in such a way that p lies on its positive side. *Precondition*: p and q are not equal.

Kernel::Plane_3 *fo.operator()(Kernel::Plane_3 h1, Kernel::Plane_3 h2)*

constructs the bisector of the two planes $h1$ and $h2$. In the general case, the bisector has a normal vector which has the same direction as the sum of the normalized normal vectors of the two planes, and passes through the intersection of *h1* and *h2*. If *h1* and *h2* are parallel, then the bisector is defined as the plane which has the same oriented normal vector as *l1*, and which is at the same distance from *h1* and *h2*. This function requires that *Kernel::RT* supports the *sqrt()* operation.

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::bisector . . . . page 696

Kernel::ConstructCartesianConstIterator_2

A model for this must provide:

Kernel::Cartesian_const_iterator_2

fo.operator()(Kernel::Point_2 p)

returns an iterator on the 0'th Cartesian coordinate of *p*.

Kernel::Cartesian_const_iterator_2

fo.operator()(Kernel::Point_2 p, int)

returns the past the end iterator of the Cartesian coordinates of *p*.

Kernel::Cartesian_const_iterator_2

fo.operator()(Kernel::Vector_2 v)

returns an iterator on the 0'th Cartesian coordinate of *v*.

Kernel::Cartesian_const_iterator_2

fo.operator()(Kernel::Vector_2 v, int)

returns the past the end iterator of the Cartesian coordinates of *v*.

Refines

AdaptableFunctor (with one argument)

See Also

Kernel::CartesianConstIterator_2 . . . . page 982

Kernel::ConstructCartesianConstIterator_3

A model for this must provide:

Kernel::Cartesian_const_iterator_3

fo.operator()(Kernel::Point_3 p)

returns an iterator on the 0'th Cartesian coordinate of *p*.

Kernel::Cartesian_const_iterator_3

fo.operator()(Kernel::Point_3 p, int)

returns the past the end iterator of the Cartesian coordinates of *p*.

Kernel::Cartesian_const_iterator_3

fo.operator()(Kernel::Vector_3 v)

returns an iterator on the 0'th Cartesian coordinate of *v*.

Kernel::Cartesian_const_iterator_3

fo.operator()(Kernel::Vector_3 v, int)

returns the past the end iterator of the Cartesian coordinates of *v*.

Refines

AdaptableFunctor (with one argument)

See Also

Kernel::CartesianConstIterator_3 . . . . page 983

Functor Concept

Kernel::ConstructCenter_2

A model for this must provide:

Kernel::Point_2	*fo.operator()(Kernel::Circle_2 c)*	
		compute the center of the circle *c*.

Refines

AdaptableFunctor (with one argument)

See Also

CGAL::Circle_2<Kernel> . . . . page 614

Kernel::ConstructCenter_3

A model for this must provide:

Kernel::Point_3	*fo.operator()(Kernel::Sphere_3 s)*	
		compute the center of the sphere *s*.
Kernel::Point_3	*fo.operator()(Kernel::Circle_3 c)*	
		compute the center of the circle *c*.

Refines

AdaptableFunctor (with one argument)

See Also

CGAL::Sphere_3<Kernel> . . . . page 666
CGAL::Circle_3<Kernel> . . . . page 663

Kernel::ConstructCentroid_2

A model for this must provide:

Kernel::Point_2 *fo.operator()(Kernel::Point_2 p, Kernel::Point_2 q, Kernel::Point_2 r)*

compute the centroid of the points p, q, and r.

Kernel::Point_2 *fo.operator()(Kernel::Point_2 p,*
Kernel::Point_2 q,
Kernel::Point_2 r,
Kernel::Point_2 s)

compute the centroid of the points p, q, r and s.

Kernel::Point_2 *fo.operator()(Kernel::Triangle_2 t)*

compute the centroid of the triangle t.

Refines

AdaptableFunctor (with three arguments)

See Also

Kernel::ConstructBarycenter_2 . . . . page 885
CGAL::centroid . . . . page 3969
CGAL::barycenter . . . . page 3971

Kernel::ConstructCentroid_3

A model for this must provide:

Kernel::Point_3 *fo.operator()(Kernel::Point_3 p, Kernel::Point_3 q, Kernel::Point_3 r)*

compute the centroid of the points p, q, and r.

Kernel::Point_3 *fo.operator()(Kernel::Point_3 p,*
Kernel::Point_3 q,
Kernel::Point_3 r,
Kernel::Point_3 s)

compute the centroid of the points p, q, r and s.

Kernel::Point_3 *fo.operator()(Kernel::Triangle_3 t)*

compute the centroid of the triangle t.

Kernel::Point_3 *fo.operator()(Kernel::Tetrahedron_3 t)*

compute the centroid of the tetrahedron t.

Refines

AdaptableFunctor (with three arguments)

See Also

Kernel::ConstructBarycenter_3 . . . . page 887
CGAL::centroid . . . . page 3969
CGAL::barycenter . . . . page 3971

Kernel::ConstructCircle_2

A model for this must provide:

Kernel::Circle_2 *fo.operator()(Kernel::Point_2 center,
Kernel::FT squared_radius,
Orientation orientation = COUNTERCLOCKWISE)*

introduces a variable of type *Kernel::Circle_2*. It is initialized to the circle with center *center*, squared radius *squared_radius* and orientation *orientation*.
Precondition: *orientation* $\neq$ *COLLINEAR*, and further, *squared_radius* ≥ 0.

Kernel::Circle_2 *fo.operator()(Kernel::Point_2 p, Kernel::Point_2 q, Kernel::Point_2 r)*

introduces a variable of type *Kernel::Circle_2*. It is initialized to the unique circle which passes through the points p, q and r. The orientation of the circle is the orientation of the point triple p, q, r.
Precondition: p, q, and r are not collinear.

Kernel::Circle_2 *fo.operator()(Kernel::Point_2 p,
Kernel::Point_2 q,
Orientation orientation = COUNTERCLOCKWISE)*

introduces a variable of type *Kernel::Circle_2*. It is initialized to the circle with diameter $\overline{pq}$ and orientation *orientation*.
Precondition: *orientation* $\neq$ *COLLINEAR*.

Kernel::Circle_2 *fo.operator()(Kernel::Point_2 center,
Orientation orientation = COUNTERCLOCKWISE)*

introduces a variable of type *Kernel::Circle_2*. It is initialized to the circle with center *center*, squared radius zero and orientation *orientation*.
Precondition: *orientation* $\neq$ *COLLINEAR*.
Postcondition: *.is_degenerate() = true*.

Refines

AdaptableFunctor (with three arguments)

See Also

CGAL::Circle_2<Kernel> . . . . page 614

Kernel::ConstructCircle_3

A model for this must provide:

Kernel::Circle_3 *fo.operator()(Kernel::Point_3 center, Kernel::FT sq_r, Kernel::Plane_3 plane)*

introduces a variable of type *Kernel::Circle_3*. It is initialized to the circle with center *center*, and squared radius *sq_r* in the plane *plane*.
Precondition: *center* lies in *plane* and $sq\_r \geq 0$.

Kernel::Circle_3 *fo.operator()(Kernel::Point_3 center, Kernel::FT sq_r, Kernel::Vector_3 n)*

introduces a variable of type *Kernel::Circle_3*. It is initialized to the circle with center *center*, and squared radius *sq_r* in the plane containing *center* and normal to *n*.
Precondition: $sq\_r \geq 0$.

Kernel::Circle_3 *fo.operator()(Kernel::Point_3 p, Kernel::Point_3 q, Kernel::Point_3 r)*

introduces a variable of type *Kernel::Point_3*. It is initialized to the circle passing through the three points.
Precondition: The three points are not collinear.

Kernel::Circle_3 *fo.operator()(Kernel::Sphere_3 sphere1, Kernel::Sphere_3 sphere2)*

introduces a variable of type *Kernel::Circle_3*. It is initialized to the circle along which the two spheres intersect.
Precondition: The two spheres intersect along a circle.

Kernel::Circle_3 *fo.operator()(Kernel::Sphere_3 sphere, Kernel::Plane_3 plane)*

introduces a variable of type *Kernel::Circle_3*. It is initialized to the circle along which the sphere and the plane intersect.
Precondition: The sphere and the plane intersect along a circle.

Kernel::Circle_3 *fo.operator()(Kernel::Plane_3 plane, Kernel::Sphere_3 sphere)*

introduces a variable of type *Kernel::Circle_3*. It is initialized to the circle along which the sphere and the plane intersect.
Precondition: The sphere and the plane intersect along a circle.

Refines

AdaptableFunctor (with three arguments)

See Also

CGAL::Circle_3<Kernel> .... page 663

Kernel::ConstructCircumcenter_2

A model for this must provide:

Kernel::Point_2 *fo.operator()(Kernel::Point_2 p, Kernel::Point_2 q)*

compute the center of the smallest circle passing through the points p and q. Note : this is the same as *Kernel::ConstructMidPoint_2*.

Kernel::Point_2 *fo.operator()(Kernel::Point_2 p, Kernel::Point_2 q, Kernel::Point_2 r)*

compute the center of the circle passing through the points p, q, and r.
Precondition: p, q, and r are not collinear.

Kernel::Point_2 *fo.operator()(Kernel::Triangle_2 t)*

compute the center of the circle passing through the three vertices of t.
Precondition: t is not degenerate.

Refines

AdaptableFunctor (with three arguments)

See Also

CGAL::circumcenter . . . . page 698

Functor Concept

Kernel::ConstructCircumcenter_3

A model for this must provide:

Kernel::Point_3 *fo.operator()(Kernel::Point_3 p, Kernel::Point_3 q)*

compute the center of the smallest circle passing through the points p and q. Note : this is the same as *Kernel::ConstructMidPoint_3*.

Kernel::Point_3 *fo.operator()(Kernel::Point_3 p, Kernel::Point_3 q, Kernel::Point_3 r, Kernel::Point_3 s)*

compute the center of the sphere passing through the points p, q, r, and s.
Precondition: p, q, r, and s are not coplanar.

Kernel::Point_3 *fo.operator()(Kernel::Tetrahedron_3 t)*

compute the center of the sphere passing through the vertices of t.
Precondition: t is not degenerate.

Kernel::Point_3 *fo.operator()(Kernel::Point_3 p, Kernel::Point_3 q, Kernel::Point_3 r)*

compute the center of the circle passing through the points p, q and r.
Precondition: p, q and r are not collinear.

Kernel::Point_3 *fo.operator()(Kernel::Triangle_3 t)*

compute the center of the circle passing through the vertices of t.
Precondition: t is not degenerate.

Refines

AdaptableFunctor (with four arguments)

See Also

CGAL::circumcenter . . . . page 698

Kernel::ConstructCrossProductVector_3

A model for this must provide:

Kernel::Vector_3 *fo.operator()(Kernel::Vector_3 v, Kernel::Vector_3 w)*

computes the cross product of v and w.

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::cross_product . . . . page 728
CGAL::determinant . . . . page 729

Kernel::ConstructDifferenceOfVectors_2

A model for this must provide:

Kernel::Vector_2 *fo.operator()(Kernel::Vector_2 v1, Kernel::Vector_2 v2)*

introduces the vector $v1 - v2$.

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::Vector_2<Kernel> . . . . page 635

Kernel::ConstructDifferenceOfVectors_3

A model for this must provide:

Kernel::Vector_3 *fo.operator()(Kernel::Vector_3 v1, Kernel::Vector_3 v2)*

introduces the vector $v1 - v2$.

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::Vector_3<Kernel> . . . . page 673

Functor
Concept

Kernel::ConstructDirection_2

A model for this must provide:

Kernel::Direction_2 *fo.operator()(Kernel::Vector_2 v)*

introduces the direction of vector *v*.

Kernel::Direction_2 *fo.operator()(Kernel::Line_2 l)*

introduces the direction of line *l*.

Kernel::Direction_2 *fo.operator()(Kernel::Ray_2 r)*

introduces the direction of ray *r*.

Kernel::Direction_2 *fo.operator()(Kernel::Segment_2 s)*

introduces the direction of segment *s*.

Refines

AdaptableFunctor (with one argument)

See Also

CGAL::Direction_2<Kernel> . . . . page 617

Kernel::ConstructDirection_3

A model for this must provide:

Kernel::Direction_3 *fo.operator()(Kernel::Vector_3 v)*

introduces a direction initialized with the direction of vector *v*.

Kernel::Direction_3 *fo.operator()(Kernel::Line_3 l)*

introduces the direction of line *l*.

Kernel::Direction_3 *fo.operator()(Kernel::Ray_3 r)*

introduces the direction of ray *r*.

Kernel::Direction_3 *fo.operator()(Kernel::Segment_3 s)*

introduces the direction of segment *s*.

Refines

AdaptableFunctor (with one argument)

See Also

CGAL::Direction_3<Kernel> . . . . page 644

Functor
Concept

Kernel::ConstructDividedVector_2

A model for this must provide:

Kernel::Vector_2	*fo.operator()(Kernel::Vector_2 v, const Kernel::RT s)*	
		introduces the vector v/s.

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::Vector_2<Kernel> . . . . page 635

Kernel::ConstructDividedVector_3

A model for this must provide:

Kernel::Vector_3	*fo.operator()(Kernel::Vector_3 v, const Kernel::RT s)*	
		introduces the vector v/s.

Functor
Concept

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::Vector_3<Kernel> . . . . page 673

Kernel::ConstructEquidistantLine_3

A model for this must provide:

Kernel::Line_3 *fo.operator()(Kernel::Point_3 p, Kernel::Point_3 q, Kernel::Point_3 r)*

constructs the line which is at the same distance from the three points p, q and r.
Precondition: p, q and r are not collinear.

Refines

AdaptableFunctor (with three arguments)

See Also

CGAL::equidistant_line . . . . page 734

Kernel::ConstructIsoCuboid_3

A model for this must provide:

Kernel::Iso_cuboid_3 *fo.operator()(Kernel::Point_3 p, Kernel::Point_3 q)*

introduces an iso-oriented cuboid with diagonal opposite vertices p and q such that p is the lexicographically smallest point in the cuboid.

Kernel::Iso_cuboid_3 *fo.operator()(Kernel::Point_3 p, Kernel::Point_3 q, int)*

introduces an iso-oriented cuboid with diagonal opposite vertices p and q. The *int* argument value is only used to distinguish the two overloaded functions.
Precondition: $p.x() <= q.x()$, $p.y() <= q.y()$ and $p.z() <= q.z()$.

Kernel::Iso_cuboid_3 *fo.operator()(Kernel::Point_3 left,
Kernel::Point_3 right,
Kernel::Point_3 bottom,
Kernel::Point_3 top,
Kernel::Point_3 far,
Kernel::Point_3 close)*

introduces an iso-oriented cuboid *fo* whose minimal x coordinate is the one of *left*, the maximal x coordinate is the one of *right*, the minimal y coordinate is the one of *bottom*, the maximal y coordinate is the one of *top*, the minimal z coordinate is the one of *far*, the maximal z coordinate is the one of *close*.

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::Iso_cuboid_3<Kernel> . . . . page 646

Kernel::ConstructIsoRectangle_2

A model for this must provide:

Kernel::Iso_rectangle_2 *fo.operator()(Kernel::Point_2 p, Kernel::Point_2 q)*

introduces an iso-oriented rectangle with diagonal opposite vertices p and q such that p is the lexicographically smallest point in the rectangle.

Kernel::Iso_rectangle_2 *fo.operator()(Kernel::Point_2 p, Kernel::Point_2 q, int)*

introduces an iso-oriented rectangle with diagonal opposite vertices p and q. The *int* argument value is only used to distinguish the two overloaded functions.
Precondition: $p.x() <= q.x()$ and $p.y() <= q.y()$.

Kernel::Iso_rectangle_2 *fo.operator()(Kernel::Point_2 left,*
Kernel::Point_2 right,
Kernel::Point_2 bottom,
Kernel::Point_2 top)

introduces an iso-oriented rectangle *fo* whose minimal x coordinate is the one of *left*, the maximal x coordinate is the one of *right*, the minimal y coordinate is the one of *bottom*, the maximal y coordinate is the one of *top*.

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::Iso_rectangle_2<Kernel> . . . . page 619

Kernel::ConstructLiftedPoint_3

A model for this must provide:

Kernel::Point_3 *fo.operator()(Kernel::Plane_3 h, Kernel::Point_2 p)*

returns a point q on plane h, such that the projection of this point onto the xy-plane is p.

Functor
Concept

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::Plane_3<Kernel> . . . . page 651

Kernel::ConstructLine_2

A model for this must provide:

Kernel::Line_2 *fo.operator()(Kernel::Point_2 p, Kernel::Point_2 q)*

introduces a line passing through the points p and q. Line is directed from p to q.

Kernel::Line_2 *fo.operator()(Kernel::Point_2 p, Kernel::Direction_2 d)*

introduces a line passing through point p with direction d.

Kernel::Line_2 *fo.operator()(Kernel::Point_2 p, Kernel::Vector_2 v)*

introduces a line passing through point p and oriented by v.

Kernel::Line_2 *fo.operator()(Kernel::Segment_2 s)*

introduces a line supporting the segment s, oriented from source to target.

Kernel::Line_2 *fo.operator()(Kernel::Ray_2 r)*

introduces a line supporting the ray r, with same orientation.

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::Line_2<Kernel> . . . . page 622

Kernel::ConstructLine_3

A model for this must provide:

Kernel::Line_3   *fo.operator()(Kernel::Point_3 p, Kernel::Point_3 q)*

introduces a line passing through the points p and q. Line is directed from p to q.

Kernel::Line_3   *fo.operator()(Kernel::Point_3 p, Kernel::Vector_3 v)*

introduces a line passing through point p and oriented by v.

Kernel::Line_3   *fo.operator()(Kernel::Point_3 p, Kernel::Direction_3 d)*

introduces a line passing through point p with direction d.

Kernel::Line_3   *fo.operator()(Kernel::Segment_3 s)*

returns the line supporting the segment s, oriented from source to target.

Kernel::Line_3   *fo.operator()(Kernel::Ray_3 r)*

returns the line supporting the ray r, with the same orientation.

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::Line_3<Kernel> . . . . page 649

Functor
Concept

Kernel::ConstructMaxVertex_2

A model for this must provide:

Kernel::Point_2 *fo.operator()(Kernel::Iso_rectangle_2 r)*

returns the vertex of *r* with lexicographically largest coordinates.

Kernel::Point_2 *fo.operator()(Kernel::Segment_2 s)*

returns the vertex of *s* with lexicographically largest coordinates.

Refines

AdaptableFunctor (with one argument)

See Also

CGAL::Iso_rectangle_2<Kernel> . . . . page 619
CGAL::Segment_2<Kernel> . . . . page 631

Kernel::ConstructMaxVertex_3

A model for this must provide:

Kernel::Point_3 *fo.operator()(Kernel::Iso_cuboid_3 c)*

returns the vertex of *c* with lexicographically largest coordinates.

Kernel::Point_3 *fo.operator()(Kernel::Segment_3 s)*

returns the vertex of *s* with lexicographically largest coordinates.

Refines

AdaptableFunctor (with one argument)

See Also

CGAL::Iso_cuboid_3<Kernel> . . . . page 646
CGAL::Segment_3<Kernel> . . . . page 661

Functor
Concept

Kernel::ConstructMidpoint_2

A model for this must provide:

Kernel::Point_2   *fo.operator()(Kernel::Point_2 p, Kernel::Point_2 q)*

computes the midpoint of the segment *pq*.

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::midpoint . . . . page 1115

Kernel::ConstructMidpoint_3

Functor
Concept

A model for this must provide:

Kernel::Point_3	*fo.operator()(Kernel::Point_3 p, Kernel::Point_3 q)*	
		computes the midpoint of the segment *pq*.

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::midpoint . . . . page 1115

Functor
Concept

Kernel::ConstructMinVertex_2

A model for this must provide:

Kernel::Point_2	*fo.operator()(Kernel::Iso_rectangle_2 r)*	
		returns the vertex of *r* with lexicographically smallest coordinates.
Kernel::Point_2	*fo.operator()(Kernel::Segment_2 s)*	
		returns the vertex of *s* with lexicographically smallest coordinates.

Refines

AdaptableFunctor (with one argument)

See Also

CGAL::Iso_rectangle_2<Kernel> . . . . page 619
CGAL::Segment_2<Kernel> . . . . page 631

Kernel::ConstructMinVertex_3

Functor
Concept

A model for this must provide:

Kernel::Point_3	*fo.operator()(Kernel::Iso_cuboid_3 c)*	
		returns the vertex of *c* with lexicographically smallest coordinates.
Kernel::Point_3	*fo.operator()(Kernel::Segment_3 s)*	
		returns the vertex of *s* with lexicographically smallest coordinates.

Refines

AdaptableFunctor (with one argument)

See Also

CGAL::Iso_cuboid_3<Kernel> . . . . page 646
CGAL::Segment_3<Kernel> . . . . page 661

Functor
Concept

Kernel::ConstructNormal_3

A model for this must provide:

Kernel::Vector_3 *fo.operator()(Kernel::Point_3 p, Kernel::Point_3 q, Kernel::Point_3 r)*

computes the normal of the vectors*q-p* and *r-p*.

Refines

AdaptableFunctor (with three arguments)

See Also

CGAL::normal . . . . page 755
CGAL::unit_normal . . . . page 777

Kernel::ConstructObject_2

Functor
Concept

A model for this must provide:

template <class T>		
Object_2	*fo.operator()(T t)*	constructs an object that contains *t* and returns it.

Refines

AdaptableFunctor (with one argument)

See Also

CGAL::Object . . . . page 4167
Kernel::Assign_2 . . . . page 817
Kernel::Assign_3 . . . . page 818
Kernel::Object_2 . . . . page 791
Kernel::Object_3 . . . . page 792

Kernel::ConstructObject_3

A model for this must provide:

template <class T>		
Object_3	*fo.operator()(T t)*	constructs an object that contains *t* and returns it.

Refines

AdaptableFunctor (with one argument)

See Also

CGAL::Object . . . . page 4167
Kernel::Assign_2 . . . . page 817
Kernel::Assign_3 . . . . page 818
Kernel::Object_2 . . . . page 791
Kernel::Object_3 . . . . page 792

Kernel::ConstructOppositeCircle_2

A model for this must provide:

Kernel::Circle_2	*fo.operator()(Kernel::Circle_2 c)*
	returns the circle with the same center and squared radius as *c*, but with opposite orientation.

Functor Concept

Refines

AdaptableFunctor (with one argument)

See Also

CGAL::Circle_2<Kernel> . . . . page 614

Kernel::ConstructOppositeDirection_2

A model for this must provide:

Kernel::Direction_2	*fo.operator()(Kernel::Direction_2 d)*	
		returns the direction opposite to d.

Refines

AdaptableFunctor (with one argument)

See Also

CGAL::Direction_2<Kernel> . . . . page 617

Kernel::ConstructOppositeDirection_3

A model for this must provide:

Kernel::Direction_3	*fo.operator()(Kernel::Direction_3 d)*
	returns the direction opposite to d.

Refines

AdaptableFunctor (with one argument)

See Also

CGAL::Direction_3<Kernel> . . . . page 644

Kernel::ConstructOppositeLine_2

A model for this must provide:

Kernel::Line_2	*fo.operator()(Kernel::Line_2 l)*	
		returns the line representing the same set of points as *l*, but with opposite direction.

Refines

AdaptableFunctor (with one argument)

See Also

CGAL::Line_2<Kernel> . . . . page 622

Kernel::ConstructOppositeLine_3

A model for this must provide:

Kernel::Line_3 *fo.operator()(Kernel::Line_3 l)*

returns the line representing the same set of points as *l*, but with opposite direction.

Functor
Concept

Refines

AdaptableFunctor (with one argument)

See Also

CGAL::Line_3<Kernel> . . . . page 649

Functor
Concept

Kernel::ConstructOppositePlane_3

A model for this must provide:

Kernel::Plane_3 *fo.operator()(Kernel::Plane_3 p)*

returns the plane representing the same set of points as p, but with opposite orientation.

Refines

AdaptableFunctor (with one argument)

See Also

CGAL::Plane_3<Kernel> . . . . page 651

Kernel::ConstructOppositeRay_2

A model for this must provide:

Kernel::Ray_2 *fo.operator()(Kernel::Ray_2 r)*

returns the ray with the same source as *r*, but in opposite direction.

Functor
Concept

Refines

AdaptableFunctor (with one argument)

See Also

CGAL::Ray_2<Kernel> . . . . page 629

Functor
Concept

Kernel::ConstructOppositeRay_3

A model for this must provide:

Kernel::Ray_3	*fo.operator()(Kernel::Ray_3 r)*	
		returns the ray with the same source as *r*, but in opposite direction.

Refines

AdaptableFunctor (with one argument)

See Also

CGAL::Ray_3<Kernel> . . . . page 659

Kernel::ConstructOppositeSegment_2

Functor Concept

A model for this must provide:

Kernel::Segment_2	*fo.operator()(Kernel::Segment_2 s)*	
		returns the segment representing the same set of points as *s*, but with opposite orientation.

Refines

AdaptableFunctor (with one argument)

See Also

CGAL::Segment_2<Kernel> . . . . page 631

Kernel::ConstructOppositeSegment_3

A model for this must provide:

Kernel::Segment_3 *fo.operator()(Kernel::Segment_3 s)*

returns the segment representing the same set of points as *s*, but with opposite orientation.

Refines

AdaptableFunctor (with one argument)

See Also

CGAL::Segment_3<Kernel> . . . . page 661

Kernel::ConstructOppositeSphere_3

A model for this must provide:

Kernel::Sphere_3 *fo.operator()(Kernel::Sphere_3 s)*

returns the sphere with the same center and squared radius as *s*, but with opposite orientation.

Refines

AdaptableFunctor (with one argument)

See Also

CGAL::Sphere_3<Kernel> . . . . page 666

Functor
Concept

Kernel::ConstructOppositeTriangle_2

A model for this must provide:

Kernel::Triangle_2 *fo.operator()(Kernel::Triangle_2 t)*

returns the triangle with opposite orientation to *t* (this flips the positive and the negative side, but not bounded and unbounded side).

Refines

AdaptableFunctor (with one argument)

See Also

CGAL::Triangle_2<Kernel> . . . . page 633

Kernel::ConstructOppositeVector_2

A model for this must provide:

Kernel::Vector_2	*fo.operator()(Kernel::Vector_2 v)*
	returns the vector *-v*.

Functor
Concept

Refines

AdaptableFunctor (with one argument)

See Also

CGAL::Vector_2<Kernel> . . . . page 635

Functor
Concept

Kernel::ConstructOppositeVector_3

A model for this must provide:

Kernel::Vector_3	*fo.operator()(Kernel::Vector_3 v)*	
		returns the vector *-v*.

Refines

AdaptableFunctor (with one argument)

See Also

CGAL::Vector_3<Kernel> . . . . page 673

Kernel::ConstructOrthogonalVector_3

A model for this must provide:

Kernel::Vector_3 *fo.operator()(Kernel::Plane_3 p)*

returns a vector that is orthogonal to the plane *p* and directed to the positive side of *p*.

Kernel::Vector_3 *fo.operator()(Kernel::Point_3 p, Kernel::Point_3 q, Kernel::Point_3 r)*

returns a vector that is orthogonal to the plane defined by *Kernel::ConstructPlane_3()(p, q, r)* and directed to the positive side of this plane.

Refines

AdaptableFunctor (with one argument)

See Also

CGAL::Plane_3<Kernel> . . . . page 651
Kernel::ConstructCrossProductVector_3 . . . . page 905

Functor
Concept

Kernel::ConstructPerpendicularDirection_2

A model for this must provide:

Kernel::Direction_2	*fo.operator()(Kernel::Direction_2 d, Orientation o)*	
		introduces a direction orthogonal to d. If o is *CLOCKWISE*, d is rotated clockwise; if o is *COUNTERCLOCKWISE*, d is rotated counterclockwise. *Precondition*: o is not *COLLINEAR*.

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::Direction_2<Kernel> . . . . page 617

Kernel::ConstructPerpendicularLine_2

A model for this must provide:

Kernel::Line_2 *fo.operator()(Kernel::Line_2 l, Kernel::Point_2 p)*

returns the line perpendicular to l and passing through p, where the direction is the direction of l rotated counterclockwise by 90 degrees.

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::Line_2<Kernel> . . . . page 622

Functor
Concept

Kernel::ConstructPerpendicularLine_3

A model for this must provide:

Kernel::Line_3 *fo.operator()(Kernel::Plane_3 pl, Kernel::Point_3 p)*

returns the line that is perpendicular to *pl* and that passes through point *p*. The line is oriented from the negative to the positive side of *pl*

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::Plane_3<Kernel> . . . . page 651

Kernel::ConstructPerpendicularPlane_3

A model for this must provide:

Kernel::Plane_3 *fo.operator()(Kernel::Line_3 l, Kernel::Point_3 p)*

returns the plane perpendicular to l passing through p, such that the normal direction of the plane coincides with the direction of the line.

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::Plane_3<Kernel> . . . . page 651

Kernel::ConstructPerpendicularVector_2

A model for this must provide:

Kernel::Vector_2	*fo.operator()(Kernel::Vector_2 v, Orientation o)*	
		returns *v* rotated clockwise by 90 degrees, if *o* is *CLOCKWISE*, and rotated counterclockwise otherwise. *Precondition*: *o* is not *COLLINEAR*.

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::Vector_2<Kernel> . . . . page 635

Kernel::ConstructPlane_3

A model for this must provide:

Kernel::Plane_3 *fo.operator()(Kernel::RT a, Kernel::RT b, Kernel::RT c, Kernel::RT d)*

creates a plane defined by the equation $ax+by+cz+d=0$. Notice that it is degenerate if $a=b=c=0$.

Kernel::Plane_3 *fo.operator()(Kernel::Point_3 p, Kernel::Point_3 q, Kernel::Point_3 r)*

creates a plane passing through the points p, q and r. The plane is oriented such that p, q and r are oriented in a positive sense (that is counterclockwise) when seen from the positive side of the plane. Notice that it is degenerate if the points are collinear.

Kernel::Plane_3 *fo.operator()(Kernel::Point_3 p, Kernel::Direction_3 d)*

introduces a plane that passes through point p and that has as an orthogonal direction equal to d.

Kernel::Plane_3 *fo.operator()(Kernel::Point_3 p, Kernel::Vector_3 v)*

introduces a plane that passes through point p and that is orthogonal to v.

Kernel::Plane_3 *fo.operator()(Kernel::Line_3 l, Kernel::Point_3 p)*

introduces a plane that is defined through the three points *l.point(0)*, *l.point(1)* and p.

Kernel::Plane_3 *fo.operator()(Kernel::Ray_3 r, Kernel::Point_3 p)*

introduces a plane that is defined through the three points *r.point(0)*, *r.point(1)* and p.

Kernel::Plane_3 *fo.operator()(Kernel::Segment_3 s, Kernel::Point_3 p)*

introduces a plane that is defined through the three points *s.source()*, *s.target()* and p.

Kernel::Plane_3 *fo.operator()(Kernel::Circle_3 c)*

introduces a plane that is defined as the plane containing the circle.

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::Plane_3<Kernel> . . . . page 651

Kernel::ConstructPointOn_2

A model for this must provide:

Kernel::Point_2 *fo.operator()(Kernel::Line_2 l, int i)*

returns an arbitrary point on *l*. It holds *point(i)* == *point(j)*, iff *i==j*. Furthermore, is directed from *point(i)* to *point(j)*, for all $i < j$.

Kernel::Point_2 *fo.operator()(Kernel::Ray_2 r, int i)*

returns a point on r. *point(0)* is the source, *point(i)*, with $i > 0$, is different from the source.
Precondition: $i \geq 0$.

Kernel::Point_2 *fo.operator()(Kernel::Segment_2 s, int i)*

returns source or target of *s*: *point(0)* returns the source of *s*, *point(1)* returns the target of *s*. The parameter *i* is taken modulo 2, which gives easy access to the other end point.

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::Line_2<Kernel> . . . . page 622
CGAL::Ray_2<Kernel> . . . . page 629
CGAL::Segment_2<Kernel> . . . . page 631

Kernel::ConstructPointOn_3

A model for this must provide:

Kernel::Point_3 *fo.operator()(Kernel::Line_3 l, int i)*

returns an arbitrary point on l. It holds *point(i)* == *point(j)*, iff *i==j*. Furthermore, is directed from *point(i)* to *point(j)*, for all $i < j$.

Kernel::Point_3 *fo.operator()(Kernel::Plane_3 h)*

returns an arbitrary point on h.

Kernel::Point_3 *fo.operator()(Kernel::Ray_3 r, int i)*

returns a point on r. *point(0)* is the source, *point(i)*, with $i > 0$, is different from the source.
Precondition: $i \geq 0$.

Kernel::Point_3 *fo.operator()(Kernel::Segment_3 s, int i)*

returns source or target of s: *point(0)* returns the source of s, *point(1)* returns the target of s. The parameter i is taken modulo 2, which gives easy access to the other end point.

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::Line_3<Kernel> . . . . page 649
CGAL::Plane_3<Kernel> . . . . page 651
CGAL::Ray_3<Kernel> . . . . page 659
CGAL::Segment_3<Kernel> . . . . page 661

Kernel::ConstructPoint_2

A model for this must provide:

Kernel::Point_2	*fo.operator()(Origin ORIGIN)*
	introduces a variable with Cartesian coordinates $(0,0)$.

Functor
Concept

Refines

AdaptableFunctor (with one argument)

See Also

CGAL::Point_2<Kernel> . . . . page 625

Kernel::ConstructPoint_3

A model for this must provide:

Kernel::Point_3 *fo.operator()(Origin ORIGIN)*

introduces a point with Cartesian coordinates $(0,0,0)$.

Refines

AdaptableFunctor (with one argument)

See Also

CGAL::Point_3<Kernel> . . . . page 655

Kernel::ConstructProjectedPoint_2

A model for this must provide:

Kernel::Point_2 *fo.operator()(Kernel::Line_2 l, Kernel::Point_2 p)*

returns the orthogonal projection of *p* onto *l*.

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::Line_2<Kernel> . . . . page 622

Kernel::ConstructProjectedPoint_3

A model for this must provide:

Kernel::Point_3 *fo.operator()(Kernel::Line_3 l, Kernel::Point_3 p)*

returns the orthogonal projection of *p* onto *l*.

Kernel::Point_3 *fo.operator()(Kernel::Plane_3 h, Kernel::Point_3 p)*

returns the orthogonal projection of *p* onto *h*.

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::Line_3<Kernel> . . . . page 649
CGAL::Plane_3<Kernel> . . . . page 651

Kernel::ConstructProjectedXYPoint_2

A model for this must provide:

Kernel::Point_2	*fo.operator()(Kernel::Plane_3 h, Kernel::Point_3 p)*	
		returns the image point of the projection of p under an affine transformation, which maps h onto the xy-plane, with the z-coordinate removed.

Functor
Concept

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::Plane_3<Kernel> . . . . page 651

Kernel::ConstructRadicalLine_2

A model for this must provide:

Kernel::Line_2 *fo.operator()(Kernel::Circle_2 c1, Kernel::Circle_2 c2)*

returns the radical line of the circles.
Precondition: The two circles don't have the same center.

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::Circle_2<Kernel> . . . . page 614

Kernel::ConstructRadicalPlane_3

A model for this must provide:

Kernel::Plane_3	*fo.operator()(Kernel::Sphere_3 sphere1, Kernel::Sphere_3 sphere2)*	
		returns the radical plane of the spheres. *Precondition*: The two spheres don't have the same center.

Refines

AdaptableFunctor (with two argument)

See Also

CGAL::Sphere_3<Kernel> . . . . page 666

Kernel::ConstructRay_2

A model for this must provide:

Kernel::Ray_2 *fo.operator()(Kernel::Point_2 p, Kernel::Point_2 q)*

introduces a ray with source p and passing through point q.

Kernel::Ray_2 *fo.operator()(Kernel::Point_2 p, Kernel::Vector_2 v)*

introduces a ray starting at source p with the direction given by v.

Kernel::Ray_2 *fo.operator()(Kernel::Point_2 p, Kernel::Direction_2 d)*

introduces a ray starting at source p with direction d.

Kernel::Ray_2 *fo.operator()(Kernel::Point_2 p, Kernel::Line_2 l)*

introduces a ray starting at source p with the same direction as l.

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::Ray_2<Kernel> . . . . page 629

Kernel::ConstructRay_3

A model for this must provide:

Kernel::Ray_3 *fo.operator()(Kernel::Point_3 p, Kernel::Point_3 q)*

introduces a ray with source p and passing through point q.

Kernel::Ray_3 *fo.operator()(Kernel::Point_3 p, Kernel::Vector_3 v)*

introduces a ray with source p and with the direction given by v.

Kernel::Ray_3 *fo.operator()(Kernel::Point_3 p, Kernel::Direction_3 d)*

introduces a ray with source p and with direction d.

Kernel::Ray_3 *fo.operator()(Kernel::Point_3 p, Kernel::Line_3 l)*

introduces a ray with source p and with the same direction as l.

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::Ray_3<Kernel> . . . . page 659

Kernel::ConstructScaledVector_2

A model for this must provide:

Kernel::Vector_2	*fo.operator()(Kernel::Vector_2 v, Kernel::RT scale)*	
		produces the vector *v* scaled by a factor *scale*.
Kernel::Vector_2	*fo.operator()(Kernel::Vector_2 v, Kernel::FT scale)*	
		produces the vector *v* scaled by a factor *scale*.

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::Vector_2<Kernel> . . . . page 635

Kernel::ConstructScaledVector_3

A model for this must provide:

Kernel::Vector_3 *fo.operator()(Kernel::Vector_3 v, Kernel::RT scale)*

produces the vector *v* scaled by a factor *scale*.

Kernel::Vector_3 *fo.operator()(Kernel::Vector_3 v, Kernel::FT scale)*

produces the vector *v* scaled by a factor *scale*.

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::Vector_3<Kernel> . . . . page 673

Kernel::ConstructSecondPoint_2

A model for this must provide:

Kernel::Point_2	*fo.operator()(Kernel::Ray_2 r)*
	returns a point different from the source on the ray *r*.

Refines

AdaptableFunctor (with one argument)

See Also

CGAL::Ray_2<Kernel> . . . . page 629

Kernel::ConstructSegment_2

A model for this must provide:

Kernel::Segment_2	*fo.operator()(Kernel::Point_2 p, Kernel::Point_2 q)*	
		introduces a segment with source p and target q. The segment is directed from the source towards the target.

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::Segment_2<Kernel> . . . . page 631

Functor
Concept

Kernel::ConstructSegment_3

A model for this must provide:

Kernel::Segment_3 *fo.operator()(Kernel::Point_3 p, Kernel::Point_3 q)*

introduces a segment with source p and target q. It is directed from the source towards the target.

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::Segment_3<Kernel> . . . . page 661

Kernel::ConstructSource_2

Functor
Concept

A model for this must provide:

Kernel::Point_2 *fo.operator()(Kernel::Segment_2 s)*

returns the source of the segment s.

Kernel::Point_2 *fo.operator()(Kernel::Ray_2 r)*

returns the source of the ray r.

Refines

AdaptableFunctor (with one argument)

See Also

CGAL::Segment_2<Kernel> . . . . page 631
CGAL::Ray_2<Kernel> . . . . page 629

Kernel::ConstructSphere_3

A model for this must provide:

Kernel::Sphere_3 *fo.operator()(Kernel::Point_3 center, Kernel::FT squared_radius, Orientation orientation = COUNTERCLOCKWISE)*

introduces a sphere initialized to the sphere with center *center*, squared radius *squared_radius* and orientation *orientation*.
Precondition: *orientation* $\neq$ *COPLANAR*, and furthermore, *squared_radius* ≥ 0.

Kernel::Sphere_3 *fo.operator()(Kernel::Point_3 p, Kernel::Point_3 q, Kernel::Point_3 r, Kernel::Point_3 s)*

introduces a sphere initialized to the unique sphere which passes through the points p, q, r and s. The orientation of the sphere is the orientation of the point quadruple p, q, r, s.
Precondition: p, q, r, and s are not coplanar.

Kernel::Sphere_3 *fo.operator()(Kernel::Point_3 p, Kernel::Point_3 q, Kernel::Point_3 r, Orientation o = COUNTERCLOCKWISE)*

introduces a sphere initialized to the smallest sphere which passes through the points p, q, and r. The orientation of the sphere is o.
Precondition: o is not *COPLANAR*.

Kernel::Sphere_3 *fo.operator()(Kernel::Point_3 p, Kernel::Point_3 q, Orientation o = COUNTERCLOCKWISE)*

introduces a sphere initialized to the smallest sphere which passes through the points p and q. The orientation of the sphere is o.
Precondition: o is not *COPLANAR*.

Kernel::Sphere_3 *fo.operator()(Kernel::Point_3 center, Orientation orientation = COUNTERCLOCKWISE)*

introduces a sphere s initialized to the sphere with center *center*, squared radius zero and orientation *orientation*.
Precondition: *orientation* $\neq$ *COPLANAR*.
Postcondition: *s.is_degenerate()* = *true*.

Kernel::Sphere_3   *fo.operator()(Kernel::Circle_3 c)*

introduces a sphere initialized to the diametral sphere of the circle.

Refines

AdaptableFunctor (with four arguments)

See Also

CGAL::Sphere_3<Kernel> . . . . page 666

Kernel::ConstructSumOfVectors_2

A model for this must provide:

Kernel::Vector_2	*fo.operator()(Kernel::Vector_2 v1, Kernel::Vector_2 v2)*
	introduces the vector $v1 + v2$.

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::Vector_2<Kernel> . . . . page 635

Kernel::ConstructSumOfVectors_3

A model for this must provide:

Kernel::Vector_3 *fo.operator()(Kernel::Vector_3 v1, Kernel::Vector_3 v2)*

introduces the vector $v1 + v2$.

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::Vector_3<Kernel> . . . . page 673

Functor
Concept

Functor
Concept

Kernel::ConstructSupportingPlane_3

A model for this must provide:

Kernel::Plane_3 *fo.operator()(Kernel::Triangle_3 t)*

returns the supporting plane of t, with same orientation.

Refines

AdaptableFunctor (with one argument)

See Also

CGAL::Triangle_3<Kernel> . . . . page 671

Kernel::ConstructTarget_2

A model for this must provide:

Kernel::Point_2 *fo.operator()(Kernel::Segment_2 s)*

returns the target of the segment *s*.

Refines

AdaptableFunctor (with one argument)

See Also

CGAL::Segment_2<Kernel> . . . . page 631

Functor
Concept

Kernel::ConstructTetrahedron_3

A model for this must provide:

Kernel::Tetrahedron_3 *fo.operator()(Kernel::Point_3 p0,*
Kernel::Point_3 p1,
Kernel::Point_3 p2,
Kernel::Point_3 p3)

introduces a tetrahedron with vertices p_0, p_1, p_2 and p_3.

Refines

AdaptableFunctor (with four arguments)

See Also

CGAL::Tetrahedron_3<Kernel> . . . . page 669

Kernel::ConstructTranslatedPoint_2

A model for this must provide:

Kernel::Point_2 *fo.operator()(Kernel::Point_2 p, Kernel::Vector_2 v)*

returns the point obtained by translating *p* by the vector *v*.

Kernel::Point_2 *fo.operator()(Origin o, Kernel::Vector_2 v)*

returns the point obtained by translating a point at the origin by the vector *v*.

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::Point_2<Kernel> . . . . page 625

Kernel::ConstructTranslatedPoint_3

A model for this must provide:

Kernel::Point_3 *fo.operator()(Kernel::Point_3 p, Kernel::Vector_3 v)*

returns the point obtained by translating p by the vector v.

Kernel::Point_3 *fo.operator()(Origin o, Kernel::Vector_3 v)*

returns the point obtained by translating a point at the origin by the vector v.

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::Point_3<Kernel> . . . . page 655

Kernel::ConstructTriangle_2

A model for this must provide:

Kernel::Triangle_2	*fo.operator()(Kernel::Point_2 p, Kernel::Point_2 q, Kernel::Point_2 r)*
	introduces a triangle with vertices p, q and r.

Functor
Concept

Refines

AdaptableFunctor (with three arguments)

See Also

CGAL::Triangle_2<Kernel> . . . . page 633

Functor
Concept

Kernel::ConstructTriangle_3

A model for this must provide:

Kernel::Triangle_3	*fo.operator()(Kernel::Point_3 p, Kernel::Point_3 q, Kernel::Point_3 r)*
	introduces a triangle with vertices p, q and r.

Refines

AdaptableFunctor (with three arguments)

See Also

CGAL::Triangle_3<Kernel> . . . . page 671

Kernel::ConstructUnitNormal_3

A model for this must provide:

Kernel::Vector_3	*fo.operator()(Kernel::Point_3 p, Kernel::Point_3 q, Kernel::Point_3 r)*	
		computes the unit normal of the vectors*q-p* and *r-p*. This requires that *Kernel::FT* supports the *sqrt* operation.

Refines

AdaptableFunctor (with three arguments)

See Also

CGAL::normal . . . . page 755
CGAL::unit_normal . . . . page 777

Kernel::ConstructVector_2

A model for this must provide:

Kernel::Vector_2 *fo.operator()(Kernel::Point_2 a, Kernel::Point_2 b)*

introduces the vector $b - a$.

Kernel::Vector_2 *fo.operator()(Origin o, Kernel::Point_2 b)*

introduces the vector b.

Kernel::Vector_2 *fo.operator()(Kernel::Point_2 a, Origin o)*

introduces the vector $-a$.

Kernel::Vector_2 *fo.operator()(Kernel::Segment_2 s)*

introduces the vector $s.target() - s.source()$.

Kernel::Vector_2 *fo.operator()(Kernel::Ray_2 r)*

introduces a vector having the same direction as r.

Kernel::Vector_2 *fo.operator()(Kernel::Line_2 l)*

introduces a vector having the same direction as l.

Kernel::Vector_2 *fo.operator()(Null_vector NULL_VECTOR)*

introduces a null vector .

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::Vector_2<Kernel> . . . . page 635
Kernel::ConstructScaledVector_2 . . . . page 960

Kernel::ConstructVector_3

Functor
Concept

A model for this must provide:

Kernel::Vector_3 *fo.operator()(Kernel::Point_3 a, Kernel::Point_3 b)*

introduces the vector $b - a$.

Kernel::Vector_3 *fo.operator()(Origin o, Kernel::Point_3 b)*

introduces the vector b.

Kernel::Vector_3 *fo.operator()(Kernel::Point_3 a, Origin o)*

introduces the vector $-a$.

Kernel::Vector_3 *fo.operator()(Kernel::Segment_3 s)*

introduces the vector $s.target() - s.source()$.

Kernel::Vector_3 *fo.operator()(Kernel::Ray_3 r)*

introduces a vector having the same direction as r.

Kernel::Vector_3 *fo.operator()(Kernel::Line_3 l)*

introduces a vector having the same direction as l.

Kernel::Vector_3 *fo.operator()(Null_vector NULL_VECTOR)*

introduces a null vector .

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::Vector_3<Kernel> . . . . page 673
Kernel::ConstructScaledVector_3 . . . . page 961

Functor
Concept

Kernel::ConstructVertex_2

A model for this must provide:

Kernel::Point_2 *fo.operator()(Kernel::Segment_2 s, int i)*

returns source or target of *s*: *fo(s,0)* returns the source of *s*, *fo(s,1)* returns the target of *s*. The parameter *i* is taken modulo 2.

Kernel::Point_2 *fo.operator()(Kernel::Iso_rectangle_2 r, int i)*

returns the i'th vertex of *r* in counterclockwise order, starting with the lower left vertex. The parameter *i* is taken modulo 4.

Kernel::Point_2 *fo.operator()(Kernel::Triangle_2 t, int i)*

returns the i'th vertex of *t*. The parameter *i* is taken modulo 3.

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::Iso_rectangle_2<Kernel> .... page 619
CGAL::Segment_2<Kernel> .... page 631
CGAL::Triangle_2<Kernel> .... page 633

Kernel::ConstructVertex_3

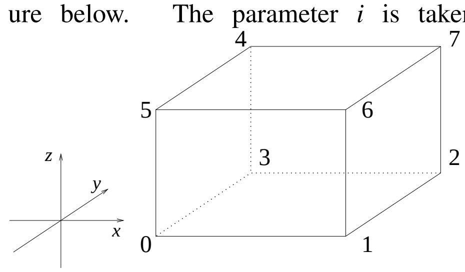
A model for this must provide:

Kernel::Point_3 *fo.operator()(Kernel::Segment_3 s, int i)*

returns source or target of *s*: *fo(s,0)* returns the source of *s*, *fo(s,1)* returns the target of *s*. The parameter *i* is taken modulo 2.

Kernel::Point_3 *fo.operator()(Kernel::Iso_cuboid_3 c, int i)*

returns the i'th vertex of *c*, as indicated in the figure below. The parameter *i* is taken modulo 8.



Kernel::Point_3 *fo.operator()(Kernel::Triangle_3 t, int i)*

returns the i'th vertex of *t*. The parameter *i* is taken modulo 3.

Kernel::Point_3 *fo.operator()(Kernel::Tetrahedron_3 t, int i)*

returns the i'th vertex of *t*. The parameter *i* is taken modulo 4.

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::Iso_cuboid_3<Kernel> .... page 646
CGAL::Segment_3<Kernel> .... page 661
CGAL::Tetrahedron_3<Kernel> .... page 669
CGAL::Triangle_3<Kernel> .... page 671

Concept

Kernel::CartesianConstIterator_2

A type representing an iterator to the Cartesian coordinates of a point in two dimensions.

Refines

CopyConstructible, Assignable, DefaultConstructible

See Also

Kernel::ConstructCartesianConstIterator_2 . . . . page 894

Kernel::CartesianConstIterator_3

A type representing an iterator to the Cartesian coordinates of a point in three dimensions.

Refines

CopyConstructible, Assignable, DefaultConstructible

See Also

Kernel::ConstructCartesianConstIterator_3 . . . . page 895

Concept

Kernel::CoplanarOrientation_3

A model for this must provide:

Orientation *fo.operator()(Kernel::Point_3 p, Kernel::Point_3 q, Kernel::Point_3 r, Kernel::Point_3 s)*

Let P be the plane defined by the points p, q, and r. Note that the order defines the orientation of P. The function computes the orientation of points p, q, and s in P: Iff p, q, s are collinear, *COLLINEAR* is returned. Iff P and the plane defined by p, q, and s have the same orientation, *POSITIVE* is returned; otherwise *NEGATIVE* is returned. *Precondition*: p, q, r, and s are coplanar and p, q, and r are not collinear.

Orientation *fo.operator()(Kernel::Point_3 p, Kernel::Point_3 q, Kernel::Point_3 r)*

If p,q,r are collinear, then *COLLINEAR* is returned. If not, then p,q,r define a plane P. The return value in this case is either *POSITIVE* or *NEGATIVE*, but we don't specify it explicitly. However, we guarantee that all calls to this predicate over 3 points in P will return a coherent orientation if considered a 2D orientation in P.

Refines

AdaptableFunctor (with four arguments)

See Also

CGAL::coplanar_orientation . . . . page 726

Kernel::CoplanarSideOfBoundedCircle_3

A model for this must provide:

Bounded_side *fo.operator()(Kernel::Point_3 p, Kernel::Point_3 q, Kernel::Point_3 r, Kernel::Point_3 s)*

returns the bounded side of the circle defined by p, q, and r on which s lies.
Precondition: p, q, r, and s are coplanar and p, q, and r are not collinear.

Refines

AdaptableFunctor (with four arguments)

See Also

CGAL::coplanar_side_of_bounded_circle . . . . page 727

Functor
Concept

Kernel::Coplanar_3

A model for this must provide:

bool	*fo.operator()(Kernel::Point_3 p, Kernel::Point_3 q, Kernel::Point_3 r, Kernel::Point_3 s)*	
		returns *true*, if p, q, r, and s are coplanar.

Refines

AdaptableFunctor (with four arguments)

See Also

CGAL::coplanar . . . . page 725

Kernel::CounterclockwiseInBetween_2

A model for this must provide:

bool *fo.operator()(Kernel::Direction_2 d, Kernel::Direction_2 d1, Kernel::Direction_2 d2)*

returns *true* iff *d* is not equal to *d1*, and while rotating counterclockwise starting at *d1*, *d* is reached strictly before *d2* is reached. Note that true is returned if *d1* == *d2*, unless also *d* == *d1*.

Refines

AdaptableFunctor (with three arguments)

See Also

CGAL::Direction_2<Kernel> . . . . page 617

Kernel::DoIntersect_2

A model for this must provide

bool	*fo.operator()(Type1 obj1, Type2 obj2)*	
		determines if two geometrical objects of type *Type1* and *Type2* intersect or not

for all pairs *Type1* and *Type2*, where the types *Type1* and *Type2* can be any of the following:

- *Kernel::Point_2*
- *Kernel::Line_2*
- *Kernel::Ray_2*
- *Kernel::Segment_2*
- *Kernel::Triangle_2*
- *Kernel::Iso_rectangle_2*

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::do_intersect . . . . page 2218

Kernel::DoIntersect_3

A model for this must provide

bool	*fo.operator()(Type1 obj1, Type2 obj2)*	
		determines if two geometrical objects of type *Type1* and *Type2* intersect or not

for all pairs *Type1* and *Type2*, where the types *Type1* and *Type2* can be any of the following:

- *Kernel::Plane_3*
- *Kernel::Line_3*
- *Kernel::Ray_3*
- *Kernel::Segment_3*
- *Kernel::Triangle_3*
- *Bbox_3*

and also for *Type1* and *Type2* of respective types

- *Kernel::Triangle_3* and *Kernel::Tetrahedron_3*
- *Kernel::Plane_3* and *Kernel::Sphere_3* (or the contrary)
- *Kernel::Sphere_3* and *Kernel::Sphere_3*.

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::do_intersect . . . . page 2218

Functor
Concept

Kernel::EqualXY_3

A model for this must provide:

bool	*fo.operator()(Kernel::Point_3 p, Kernel::Point_3 q)*	
		returns true iff p and q have the same Cartesian x-coordinate and the same Cartesian y-coordinate.

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::compare_xy . . . . page 714

Kernel::EqualX_2

A model for this must provide:

bool *fo.operator()(Kernel::Point_2 p, Kernel::Point_2 q)*

returns true iff p and q have the same Cartesian x-coordinate.

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::x_equal . . . . page 779

Functor
Concept

Functor
Concept

Kernel::EqualX_3

A model for this must provide:

bool *fo.operator()(Kernel::Point_3 p, Kernel::Point_3 q)*

returns true iff p and q have the same Cartesian x-coordinate.

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::x_equal . . . . page 779

Kernel::EqualY_2

A model for this must provide:

bool	*fo.operator()(Kernel::Point_2 p, Kernel::Point_2 q)*	
		returns true iff p and q have the same Cartesian y-coordinate.

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::y_equal . . . . page 780

Kernel::EqualY_3

A model for this must provide:

bool	*fo.operator()(Kernel::Point_3 p, Kernel::Point_3 q)*	
		returns true iff p and q have the same Cartesian y-coordinate.

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::y_equal .... page 780

Kernel::EqualZ_3

A model for this must provide:

bool	*fo.operator()(Kernel::Point_3 p, Kernel::Point_3 q)*	
		returns true iff p and q have the same Cartesian z-coordinate.

Functor Concept

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::z_equal . . . . page 781

Kernel::Equal_2

A model for this must provide the following operations. For all of them *fo(x,y)* returns true iff *x* and *y* are equal.

bool	*fo.operator()(Kernel::Point_2 x, Kernel::Point_2 y)*
bool	*fo.operator()(Kernel::Vector_2 x, Kernel::Vector_2 y)*
bool	*fo.operator()(Kernel::Direction_2 x, Kernel::Direction_2 y)*
bool	*fo.operator()(Kernel::Line_2 x, Kernel::Line_2 y)*
bool	*fo.operator()(Kernel::Ray_2 x, Kernel::Ray_2 y)*
bool	*fo.operator()(Kernel::Segment_2 x, Kernel::Segment_2 y)*
bool	*fo.operator()(Kernel::Circle_2 x, Kernel::Circle_2 y)*
bool	*fo.operator()(Kernel::Triangle_2 x, Kernel::Triangle_2 y)*
bool	*fo.operator()(Kernel::Iso_rectangle_2 x, Kernel::Iso_rectangle_2 y)*

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::Circle_2<Kernel> . . . . page 614
CGAL::Direction_2<Kernel> . . . . page 617
CGAL::Iso_rectangle_2<Kernel> . . . . page 619
CGAL::Line_2<Kernel> . . . . page 622
CGAL::Point_2<Kernel> . . . . page 625
CGAL::Ray_2<Kernel> . . . . page 629
CGAL::Segment_2<Kernel> . . . . page 631
CGAL::Triangle_2<Kernel> . . . . page 633
CGAL::Vector_2<Kernel> . . . . page 635

Kernel::Equal_3

A model for this must provide the following operations. For all of them *fo(x,y)* returns true iff *x* and *y* are equal.

bool	*fo.operator()(Kernel::Point_3 x, Kernel::Point_3 y)*
bool	*fo.operator()(Kernel::Vector_3 x, Kernel::Vector_3 y)*
bool	*fo.operator()(Kernel::Direction_3 x, Kernel::Direction_3 y)*
bool	*fo.operator()(Kernel::Line_3 x, Kernel::Line_3 y)*
bool	*fo.operator()(Kernel::Plane_3 x, Kernel::Plane_3 y)*
bool	*fo.operator()(Kernel::Ray_3 x, Kernel::Ray_3 y)*
bool	*fo.operator()(Kernel::Segment_3 x, Kernel::Segment_3 y)*
bool	*fo.operator()(Kernel::Circle_3 x, Kernel::Circle_3 y)*
bool	*fo.operator()(Kernel::Sphere_3 x, Kernel::Sphere_3 y)*
bool	*fo.operator()(Kernel::Triangle_3 x, Kernel::Triangle_3 y)*
bool	*fo.operator()(Kernel::Tetrahedron_3 x, Kernel::Tetrahedron_3 y)*
bool	*fo.operator()(Kernel::Iso_cuboid_3 x, Kernel::Iso_cuboid_3 y)*

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::Direction_3<Kernel> .... page 644
CGAL::Iso_cuboid_3<Kernel> .... page 646
CGAL::Line_3<Kernel> .... page 649
CGAL::Plane_3<Kernel> .... page 651
CGAL::Point_3<Kernel> .... page 655
CGAL::Ray_3<Kernel> .... page 659
CGAL::Segment_3<Kernel> .... page 661
CGAL::Sphere_3<Kernel> .... page 666
CGAL::Tetrahedron_3<Kernel> .... page 669
CGAL::Triangle_3<Kernel> .... page 671
CGAL::Vector_3<Kernel> .... page 673

Functor
Concept

Functor
Concept

Kernel::HasOnBoundary_2

A model for this must provide:

bool *fo.operator()(Kernel::Circle_2 c, Kernel::Point_2 p)*

returns true iff p lies on the boundary of c.

bool *fo.operator()(Kernel::Iso_rectangle_2 i, Kernel::Point_2 p)*

returns true iff p lies on the boundary of i.

bool *fo.operator()(Kernel::Triangle_2 t, Kernel::Point_2 p)*

returns true iff p lies on the boundary of t.

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::Circle_2<Kernel> . . . . page 614
CGAL::Iso_rectangle_2<Kernel> . . . . page 619
CGAL::Triangle_2<Kernel> . . . . page 633

Kernel::HasOnBoundary_3

A model for this must provide:

bool	*fo.operator()(Kernel::Sphere_3 s, Kernel::Point_3 p)*	
		returns true iff p lies on the boundary of s.
bool	*fo.operator()(Kernel::Tetrahedron_3 t, Kernel::Point_3 p)*	
		returns true iff p lies on the boundary of t.
bool	*fo.operator()(Kernel::Iso_cuboid_3 c, Kernel::Point_3 p)*	
		returns true iff p lies on the boundary of c.

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::Iso_cuboid_3<Kernel> . . . . page 646
CGAL::Sphere_3<Kernel> . . . . page 666
CGAL::Tetrahedron_3<Kernel> . . . . page 669

Kernel::HasOnBoundedSide_2

A model for this must provide:

bool *fo.operator()(Kernel::Circle_2 c, Kernel::Point_2 p)*

returns true iff p lies on the bounded side of c.

bool *fo.operator()(Kernel::Iso_rectangle_2 i, Kernel::Point_2 p)*

returns true iff p lies on the bounded side of i.

bool *fo.operator()(Kernel::Triangle_2 t, Kernel::Point_2 p)*

returns true iff p lies on the bounded side of t.

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::Circle_2<Kernel> . . . . page 614
CGAL::Iso_rectangle_2<Kernel> . . . . page 619
CGAL::Triangle_2<Kernel> . . . . page 633

Kernel::HasOnBoundedSide_3

A model for this must provide:

bool *fo.operator()(Kernel::Sphere_3 s, Kernel::Point_3 p)*

returns true iff p lies on the bounded side of s.

bool *fo.operator()(Kernel::Tetrahedron_3 t, Kernel::Point_3 p)*

returns true iff p lies on the bounded side of t.

bool *fo.operator()(Kernel::Iso_cuboid_3 c, Kernel::Point_3 p)*

returns true iff p lies on the bounded side of c.

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::Iso_cuboid_3<Kernel> . . . . page 646
CGAL::Sphere_3<Kernel> . . . . page 666
CGAL::Tetrahedron_3<Kernel> . . . . page 669

Kernel::HasOnNegativeSide_2

A model for this must provide:

bool *fo.operator()(Kernel::Circle_2 c, Kernel::Point_2 p)*

returns true iff p lies on the negative side of c.

bool *fo.operator()(Kernel::Line_2 l, Kernel::Point_2 p)*

returns true iff p lies on the negative side of l (l is considered a half-space).

bool *fo.operator()(Kernel::Triangle_2 t, Kernel::Point_2 p)*

returns true iff p lies on the negative side of t.

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::Circle_2<Kernel> . . . . page 614
CGAL::Line_2<Kernel> . . . . page 622
CGAL::Triangle_2<Kernel> . . . . page 633

Kernel::HasOnNegativeSide_3

A model for this must provide:

bool	*fo.operator()(Kernel::Plane_3 h, Kernel::Point_3 p)*	
		returns true iff p lies on the negative side of h (h is considered a half-space).
bool	*fo.operator()(Kernel::Sphere_3 s, Kernel::Point_3 p)*	
		returns true iff p lies on the negative side of s.
bool	*fo.operator()(Kernel::Tetrahedron_3 t, Kernel::Point_3 p)*	
		returns true iff p lies on the negative side of t.

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::Plane_3<Kernel> . . . . page 651
CGAL::Sphere_3<Kernel> . . . . page 666
CGAL::Tetrahedron_3<Kernel> . . . . page 669

Kernel::HasOnPositiveSide_2

A model for this must provide:

bool	*fo.operator()(Kernel::Circle_2 c, Kernel::Point_2 p)*	
		returns true iff p lies on the positive side of c.
bool	*fo.operator()(Kernel::Line_2 l, Kernel::Point_2 p)*	
		returns true iff p lies on the positive side of l (l is considered a half-space).
bool	*fo.operator()(Kernel::Triangle_2 t, Kernel::Point_2 p)*	
		returns true iff p lies on the positive side of t.

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::Circle_2<Kernel> . . . . page 614
CGAL::Line_2<Kernel> . . . . page 622
CGAL::Triangle_2<Kernel> . . . . page 633

Kernel::HasOnPositiveSide_3

A model for this must provide:

bool	*fo.operator()(Kernel::Plane_3 h, Kernel::Point_3 p)*	
		returns true iff p lies on the positive side of h (h is considered a half-space).
bool	*fo.operator()(Kernel::Sphere_3 s, Kernel::Point_3 p)*	
		returns true iff p lies on the positive side of s.
bool	*fo.operator()(Kernel::Tetrahedron_3 t, Kernel::Point_3 p)*	
		returns true iff p lies on the positive side of t.

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::Plane_3<Kernel> . . . . page 651
CGAL::Sphere_3<Kernel> . . . . page 666
CGAL::Tetrahedron_3<Kernel> . . . . page 669

Kernel::HasOnUnboundedSide_2

A model for this must provide:

bool *fo.operator()(Kernel::Circle_2 c, Kernel::Point_2 p)*

returns true iff p lies on the unbounded side of c.

bool *fo.operator()(Kernel::Iso_rectangle_2 i, Kernel::Point_2 p)*

returns true iff p lies on the unbounded side of i.

bool *fo.operator()(Kernel::Triangle_2 t, Kernel::Point_2 p)*

returns true iff p lies on the unbounded side of t.

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::Circle_2<Kernel> . . . . page 614
CGAL::Iso_rectangle_2<Kernel> . . . . page 619
CGAL::Triangle_2<Kernel> . . . . page 633

Kernel::HasOnUnboundedSide_3

A model for this must provide:

bool *fo.operator()(Kernel::Sphere_3 s, Kernel::Point_3 p)*

returns true iff p lies on the unbounded side of s.

bool *fo.operator()(Kernel::Tetrahedron_3 t, Kernel::Point_3 p)*

returns true iff p lies on the unbounded side of t.

bool *fo.operator()(Kernel::Iso_cuboid_3 c, Kernel::Point_3 p)*

returns true iff p lies on the unbounded side of c.

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::Iso_cuboid_3<Kernel> . . . . page 646
CGAL::Sphere_3<Kernel> . . . . page 666
CGAL::Tetrahedron_3<Kernel> . . . . page 669

Kernel::HasOn_2

A model for this must provide:

bool *fo.operator()(Kernel::Line_2 l, Kernel::Point_2 p)*

returns true iff p lies on l.

bool *fo.operator()(Kernel::Ray_2 r, Kernel::Point_2 p)*

returns true iff p lies on r.

bool *fo.operator()(Kernel::Segment_2 s, Kernel::Point_2 p)*

returns true iff p lies on s.

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::Line_2<Kernel> . . . . page 622
CGAL::Ray_2<Kernel> . . . . page 629
CGAL::Segment_2<Kernel> . . . . page 631

Kernel::HasOn_3

A model for this must provide:

bool *fo.operator()(Kernel::Circle_3 c, Kernel::Point_3 p)*

returns true iff p lies on c.

bool *fo.operator()(Kernel::Line_3 l, Kernel::Point_3 p)*

returns true iff p lies on l.

bool *fo.operator()(Kernel::Ray_3 r, Kernel::Point_3 p)*

returns true iff p lies on r.

bool *fo.operator()(Kernel::Segment_3 s, Kernel::Point_3 p)*

returns true iff p lies on s.

bool *fo.operator()(Kernel::Plane_3 pl, Kernel::Point_3 p)*

returns true iff p lies on pl.

bool *fo.operator()(Kernel::Plane_3 pl, Kernel::Line_3 l)*

returns true iff l lies on pl.

bool *fo.operator()(Kernel::Plane_3 pl, Kernel::Circle_3 c)*

returns true iff c lies on pl.

bool *fo.operator()(Kernel::Sphere_3 s, Kernel::Point_3 c)*

returns true iff c lies on s.

bool *fo.operator()(Kernel::Sphere_3 s, Kernel::Circle_3 c)*

returns true iff c lies on s.

bool *fo.operator()(Kernel::Triangle_3 t, Kernel::Point_3 p)*

returns true iff p lies on t.

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::Circle_3<Kernel> .... page 663
CGAL::Line_3<Kernel> .... page 649
CGAL::Plane_3<Kernel> .... page 651
CGAL::Point_3<Kernel> .... page 655
CGAL::Ray_3<Kernel> .... page 659
CGAL::Segment_3<Kernel> .... page 661
CGAL::Sphere_3<Kernel> .... page 666
CGAL::Triangle_3<Kernel> .... page 671

Kernel::Intersect_2

A model for this must provide

Kernel::Object_2	*fo.operator()(Type1 obj1, Type2 obj2)*	
		computes the intersection region of two geometrical objects of type *Type1* and *Type2*

for all pairs *Type1* and *Type2*, where the types *Type1* and *Type2* can be any of the following:

- *Kernel::Line_2*
- *Kernel::Ray_2*
- *Kernel::Segment_2*
- *Kernel::Triangle_2*
- *Kernel::Iso_rectangle_2*

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::intersection . . . . page 1553

Functor
Concept

Functor
Concept

Kernel::Intersect_3

A model for this must provide

Kernel::Object_3   *fo.operator()(Type1 obj1, Type2 obj2)*

computes the intersection region of two geometrical objects of type *Type1* and *Type2*

for all pairs *Type1* and *Type2*, where the types *Type1* and *Type2* can be any of the following:

- *Kernel::Plane_3*
- *Kernel::Line_3*
- *Kernel::Ray_3*
- *Kernel::Segment_3*
- *Kernel::Triangle_3*

Kernel::Object_3   *fo.operator()(Kernel::Sphere_3 s1, Kernel::Sphere_3 s2)*

computes the intersection of two spheres. The result can be either a *Kernel::Point_3*, a *Kernel::Circle_3*, a *Kernel::Sphere_3* or empty.

Kernel::Object_3   *fo.operator()(Kernel::Plane_3 p, Kernel::Sphere_3 s)*

computes the intersection of a plane and a sphere. The result can be either a *Kernel::Point_3*, a *Kernel::Circle_3* or empty.

Kernel::Object_3   *fo.operator()(Kernel::Sphere_3 s, Kernel::Plane_3 p)*

computes the intersection of a plane and a sphere. The result can be either a *Kernel::Point_3*, a *Kernel::Circle_3* or empty.

Kernel::Object_3   *fo.operator()(Kernel::Plane_3 pl1, Kernel::Plane_3 pl2, Kernel::Plane_3 pl3)*

computes the intersection of three planes. The result can be either a *Kernel::Point_3*, a *Kernel::Line_3*, a *Kernel::Plane_3*, or empty.

Refines

AdaptableFunctor (with two or three arguments)

See Also

CGAL::intersection . . . . page 1553

Functor
Concept

Kernel::IsDegenerate_2

A model for this must provide:

bool *fo.operator()(Kernel::Circle_2 o)*

returns true iff *o* is degenerate.

bool *fo.operator()(Kernel::Iso_rectangle_2 o)*

returns true iff *o* is degenerate.

bool *fo.operator()(Kernel::Line_2 o)*

returns true iff *o* is degenerate.

bool *fo.operator()(Kernel::Ray_2 o)*

returns true iff *o* is degenerate.

bool *fo.operator()(Kernel::Segment_2 o)*

returns true iff *o* is degenerate.

bool *fo.operator()(Kernel::Triangle_2 o)*

returns true iff *o* is degenerate.

Refines

AdaptableFunctor (with one argument)

See Also

CGAL::Circle_2<Kernel> . . . . page 614
CGAL::Iso_rectangle_2<Kernel> . . . . page 619
CGAL::Line_2<Kernel> . . . . page 622
CGAL::Ray_2<Kernel> . . . . page 629
CGAL::Segment_2<Kernel> . . . . page 631
CGAL::Triangle_2<Kernel> . . . . page 633

Kernel::IsDegenerate_3

A model for this must provide:

bool	*fo.operator()(Kernel::Circle_3 o)*	
		returns true iff *o* is degenerate.
bool	*fo.operator()(Kernel::Iso_cuboid_3 o)*	
		returns true iff *o* is degenerate.
bool	*fo.operator()(Kernel::Line_3 o)*	
		returns true iff *o* is degenerate.
bool	*fo.operator()(Kernel::Plane_3 o)*	
		returns true iff *o* is degenerate.
bool	*fo.operator()(Kernel::Ray_3 o)*	
		returns true iff *o* is degenerate.
bool	*fo.operator()(Kernel::Segment_3 o)*	
		returns true iff *o* is degenerate.
bool	*fo.operator()(Kernel::Sphere_3 o)*	
		returns true iff *o* is degenerate.
bool	*fo.operator()(Kernel::Tetrahedron_3 o)*	
		returns true iff *o* is degenerate.
bool	*fo.operator()(Kernel::Triangle_3 o)*	
		returns true iff *o* is degenerate.

Refines

AdaptableFunctor (with one argument)

See Also

CGAL::Circle_3<Kernel> . . . . page 663
CGAL::Iso_cuboid_3<Kernel> . . . . page 646
CGAL::Line_3<Kernel> . . . . page 649
CGAL::Plane_3<Kernel> . . . . page 651
CGAL::Point_3<Kernel> . . . . page 655
CGAL::Ray_3<Kernel> . . . . page 659
CGAL::Segment_3<Kernel> . . . . page 661
CGAL::Sphere_3<Kernel> . . . . page 666
CGAL::Tetrahedron_3<Kernel> . . . . page 669
CGAL::Triangle_3<Kernel> . . . . page 671

Kernel::IsHorizontal_2

A model for this must provide:

bool *fo.operator()(Kernel::Line_2 o)*

returns true iff *o* is horizontal.

bool *fo.operator()(Kernel::Ray_2 o)*

returns true iff *o* is horizontal.

bool *fo.operator()(Kernel::Segment_2 o)*

returns true iff *o* is horizontal.

Refines

AdaptableFunctor (with one argument)

See Also

CGAL::Line_2<Kernel> .... page 622
CGAL::Ray_2<Kernel> .... page 629
CGAL::Segment_2<Kernel> .... page 631

Kernel::IsVertical_2

A model for this must provide:

bool *fo.operator()(Kernel::Line_2 o)*

returns true iff *o* is vertical.

bool *fo.operator()(Kernel::Ray_2 o)*

returns true iff *o* is vertical.

bool *fo.operator()(Kernel::Segment_2 o)*

returns true iff *o* is vertical.

Refines

AdaptableFunctor (with one argument)

See Also

CGAL::Line_2<Kernel> . . . . page 622
CGAL::Ray_2<Kernel> . . . . page 629
CGAL::Segment_2<Kernel> . . . . page 631

Kernel::LeftTurn_2

A model for this must provide:

bool *fo.operator()(Kernel::Point_2 p, Kernel::Point_2 q, Kernel::Point_2 r)*

returns *true*, iff the three points p, q and r form a left turn.

Refines

AdaptableFunctor (with three arguments)

See Also

CGAL::left_turn . . . . page 746

Functor Concept

Kernel::LessDistanceToPoint_2

A model for this must provide:

bool	*fo.operator()(Kernel::Point_2 p, Kernel::Point_2 q, Kernel::Point_2 r)*	
		returns true iff the distance of q to p is smaller than the distance of r to p.

Refines

AdaptableFunctor (with three arguments)

See Also

CGAL::has_smaller_distance_to_point . . . . page 738

Kernel::LessDistanceToPoint_3

Functor Concept

A model for this must provide:

bool	*fo.operator()(Kernel::Point_3 p, Kernel::Point_3 q, Kernel::Point_3 r)*	
		returns true iff the distance of q to p is smaller than the distance of r to p.

Refines

AdaptableFunctor (with three arguments)

See Also

CGAL::has_smaller_distance_to_point . . . . page 738

Functor
Concept

Kernel::LessRotateCCW_2

A model for this must provide:

bool	*fo.operator()(Kernel::Point_2 p, Kernel::Point_2 q, Kernel::Point_2 r)*	
		returns true iff the three points p, q and r form a left turn or if they are collinear and the distance of q to p is larger than the distance of r to p, where p is the point passed to the object at construction. *Precondition*: p does not lie in the interior of the segment rq, i.e. p is an extreme point with respect to $\{p,q,r\}$.

Refines

AdaptableFunctor (with three arguments)

Kernel::LessSignedDistanceToLine_2

A model for this must provide:

bool *fo.operator()(Kernel::Line_2 l, Kernel::Point_2 p, Kernel::Point_2 q)*

returns *true* if the signed distance from p and the oriented line l is smaller than the signed distance of q and l.

bool *fo.operator()(Kernel::Point_2 p, Kernel::Point_2 q, Kernel::Point_2 r, Kernel::Point_2 s)*

returns *true* if the signed distance from r and the oriented line l defined by p and q is smaller than the signed distance of s and l.
Precondition: $p! = q$.

Refines

AdaptableFunctor (with four arguments)

See Also

CGAL::has_smaller_signed_distance_to_line . . . . page 739

Functor
Concept

Functor
Concept

Kernel::LessSignedDistanceToPlane_3

A model for this must provide:

bool *fo.operator()(Kernel::Plane_3 p, Kernel::Point_3 q, Kernel::Point_3 r)*

returns true, iff the signed distance from point q to plane p is smaller than the signed distance from point r to p.

bool *fo.operator()(Kernel::Point_3 p1, Kernel::Point_3 p2, Kernel::Point_3 p3, Kernel::Point_3 q, Kernel::Point_3 r)*

returns true, iff the signed distance from point q to the plane p defined by *p1, p2, p3* is smaller than the signed distance from point r to p.
Precondition: p, q, and r are not collinear.

Refines

AdaptableFunctor (with three arguments)

See Also

CGAL::has_smaller_signed_distance_to_plane . . . . page 740

Kernel::LessXYZ_3

A model for this must provide:

bool *fo.operator()(Kernel::Point_3 p, Kernel::Point_3 q)*

returns true iff the x-coordinate of p is smaller than the x-coordinate of q or if the are the same and the y-coordinate of p is smaller than the y-coordinate of q, or, if both x- and y- coordinate are identical and the z-coordinate of p is smaller than the z-coordinate of q.

Functor Concept

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::lexicographically_xyz_smaller . . . . page 747

Kernel::LessXY_2

A model for this must provide:

bool *fo.operator()(Kernel::Point_2 p, Kernel::Point_2 q)*

returns true iff the x-coordinate of p is smaller than the x-coordinate of q or if the are the same and the y-coordinate of p is smaller than the y-coordinate of q.

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::lexicographically_xy_smaller . . . . page 751

Kernel::LessXY_3

A model for this must provide:

bool *fo.operator()(Kernel::Point_3 p, Kernel::Point_3 q)*

returns true iff the x-coordinate of p is smaller than the x-coordinate of q or if the are the same and the y-coordinate of p is smaller than the y-coordinate of q.

Functor
Concept

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::compare_xy . . . . page 714

Functor
Concept

Kernel::LessX_2

A model for this must provide:

bool	*fo.operator()(Kernel::Point_2 p, Kernel::Point_2 q)*	
		returns true iff the x-coordinate of p is smaller than the x-coordinate of q.

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::compare_x . . . . page 712

Kernel::LessX_3

A model for this must provide:

bool *fo.operator()(Kernel::Point_3 p, Kernel::Point_3 q)*

returns true iff the x-coordinate of p is smaller than the x-coordinate of q.

Functor
Concept

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::compare_x . . . . page 712

Functor
Concept

Kernel::LessYX_2

A model for this must provide:

bool *fo.operator()(Kernel::Point_2 p, Kernel::Point_2 q)*

returns true iff the y-coordinate of p is smaller than the y-coordinate of q or if the are the same and the x-coordinate of p is smaller than the x-coordinate of q.

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::compare_yx . . . . page 723

Kernel::LessY_2

Functor Concept

A model for this must provide:

bool	*fo.operator()(Kernel::Point_2 p, Kernel::Point_2 q)*	
		returns true iff the y-coordinate of p is smaller than the y-coordinate of q.

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::compare_y . . . . page 719

Kernel::LessY_3

A model for this must provide:

bool *fo.operator()(Kernel::Point_3 p, Kernel::Point_3 q)*

returns true iff the y-coordinate of p is smaller than the y-coordinate of q.

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::compare_y . . . . page 719

Kernel::LessZ_3

A model for this must provide:

bool *fo.operator()(Kernel::Point_3 p, Kernel::Point_3 q)*

returns true iff the z-coordinate of p is smaller than the z-coordinate of q.

Functor
Concept

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::compare_z . . . . page 724

Kernel::Orientation_2

A model for this must provide:

Orientation *fo.operator()(Kernel::Point_2 p, Kernel::Point_2 q, Kernel::Point_2 r)*

returns *LEFT_TURN*, if r lies to the left of the oriented line l defined by p and q, returns *RIGHT_TURN* if r lies to the right of l, and returns *COLLINEAR* if r lies on l.

Orientation *fo.operator()(Kernel::Vector_2 u, Kernel::Vector_2 v)*

returns *LEFT_TURN* if u and v form a left turn, returns *RIGHT_TURN* if u and v form a right turn, and returns *COLLINEAR* if u and v are collinear.

Refines

AdaptableFunctor (with three arguments)

See Also

CGAL::orientation . . . . page 1116

Kernel::Orientation_3

A model for this must provide:

Orientation *fo.operator()(Kernel::Point_3 p, Kernel::Point_3 q, Kernel::Point_3 r, Kernel::Point_3 s)*

returns *POSITIVE*, if s lies on the positive side of the oriented plane h defined by p, q, and r, returns *NEGATIVE* if s lies on the negative side of h, and returns *COPLANAR* if s lies on h.

Orientation *fo.operator()(Kernel::Vector_3 u, Kernel::Vector_3 v, Kernel::Vector_3 w)*

returns *POSITIVE* if u, v and w are positively oriented, returns *NEGATIVE* if u, v and w are negatively oriented, and returns *COPLANAR* if u, v and w are coplanar.

Orientation *fo.operator()(Kernel::Sphere_3 s)*

returns the orientation of the sphere s.

Refines

AdaptableFunctor (with four arguments)

See Also

CGAL::orientation . . . . page 1116

Kernel::OrientedSide_2

A model for this must provide:

Oriented_side *fo.operator()(Kernel::Circle_2 c, Kernel::Point_2 p)*

returns *ON_ORIENTED_BOUNDARY*, *ON_NEGATIVE_SIDE*, or the constant *ON_POSITIVE_SIDE*, depending on the position of p relative to the oriented circle c.

Oriented_side *fo.operator()(Kernel::Line_2 l, Kernel::Point_2 p)*

returns *ON_ORIENTED_BOUNDARY*, *ON_NEGATIVE_SIDE*, or the constant *ON_POSITIVE_SIDE*, depending on the position of p relative to the oriented line l.

Oriented_side *fo.operator()(Kernel::Triangle_2 t, Kernel::Point_2 p)*

returns *ON_ORIENTED_BOUNDARY*, *ON_NEGATIVE_SIDE*, or the constant *ON_POSITIVE_SIDE*, depending on the position of p relative to the oriented triangle t.

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::Circle_2<Kernel> . . . . page 614
CGAL::Line_2<Kernel> . . . . page 622
CGAL::Triangle_2<Kernel> . . . . page 633

Kernel::OrientedSide_3

A model for this must provide:

Oriented_side *fo.operator()(Kernel::Plane_3 h, Kernel::Point_3 p)*

returns *ON_ORIENTED_BOUNDARY*, *ON_NEGATIVE_SIDE*, or the constant *ON_POSITIVE_SIDE*, depending on the position of p relative to the oriented plane h.

Oriented_side *fo.operator()(Kernel::Tetrahedron_3 t, Kernel::Point_3 p)*

returns *ON_ORIENTED_BOUNDARY*, *ON_NEGATIVE_SIDE*, or the constant *ON_POSITIVE_SIDE*, depending on the position of p relative to the oriented tetrahedron t.

Oriented_side *fo.operator()(Kernel::Sphere_3 s, Kernel::Point_3 p)*

returns *ON_ORIENTED_BOUNDARY*, *ON_NEGATIVE_SIDE*, or the *ON_POSITIVE_SIDE*, depending on the position of p relative to the oriented sphere s.

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::Plane_3<Kernel> . . . . page 651
CGAL::Sphere_3<Kernel> . . . . page 666
CGAL::Tetrahedron_3<Kernel> . . . . page 669

Functor Concept

Kernel::SideOfBoundedCircle_2

A model for this must provide:

Bounded_side *fo.operator()(Kernel::Point_2 p, Kernel::Point_2 q, Kernel::Point_2 r, Kernel::Point_2 t)*

returns the relative position of point t to the circle defined by p, q and r. The order of the points p, q and r does not matter.
Precondition: p, q and r are not collinear.

Bounded_side *fo.operator()(Kernel::Point_2 p, Kernel::Point_2 q, Kernel::Point_2 t)*

returns the position of the point t relative to the circle that has pq as its diameter.

Refines

AdaptableFunctor (with four arguments)

See Also

CGAL::side_of_bounded_circle . . . . page 769

Kernel::SideOfBoundedSphere_3

A model for this must provide:

Bounded_side *fo.operator()(Kernel::Point_3 p, Kernel::Point_3 q, Kernel::Point_3 r, Kernel::Point_3 s, Kernel::Point_3 t)*

returns the relative position of point t to the sphere defined by p, q, r, and s. The order of the points p, q, r, and s does not matter.
Precondition: p, q, r and s are not coplanar.

Bounded_side *fo.operator()(Kernel::Point_3 p, Kernel::Point_3 q, Kernel::Point_3 r, Kernel::Point_3 t)*

returns the position of the point t relative to the sphere passing through p, q, and r and whose center is in the plane defined by these three points.

Bounded_side *fo.operator()(Kernel::Point_3 p, Kernel::Point_3 q, Kernel::Point_3 t)*

returns the position of the point t relative to the sphere that has pq as its diameter.

Refines

AdaptableFunctor (with five arguments)

See Also

CGAL::side_of_bounded_sphere . . . . page 1118

Functor
Concept

Kernel::SideOfOrientedCircle_2

A model for this must provide:

Oriented_side *fo.operator()(Kernel::Point_2 p, Kernel::Point_2 q, Kernel::Point_2 r, Kernel::Point_2 t)*

returns the relative position of point t to the oriented circle defined by p, q and r. The order of the points p, q and r is important, since it determines the orientation of the implicitly constructed circle.
If p, q and r are collinear, the circle degenerates in a line. *ON_ORIENTED_BOUNDARY* is returned if t is also collinear or if two points are identical, otherwise, *side_of_oriented_circle(r, q, t, p)* is returned.

Refines

AdaptableFunctor (with four arguments)

See Also

CGAL::side_of_oriented_circle . . . . page 771

Kernel::SideOfOrientedSphere_3

A model for this must provide:

Oriented_side	*fo.operator()(Kernel::Point_3 p, Kernel::Point_3 q, Kernel::Point_3 r, Kernel::Point_3 s, Kernel::Point_3 t)*	

returns the relative position of point t to the oriented sphere defined by p, q, r and s. The order of the points p, q, r, and s is important, since it determines the orientation of the implicitly constructed sphere. If the points p, q, r and s are positive oriented, positive side is the bounded interior of the sphere.
In case of degeneracies, *ON_ORIENTED_BOUNDARY* is returned if all points are coplanar. Otherwise, there is a cyclic permutation of the five points that puts four non coplanar points first, it is used to answer the predicate: e.g. *side_of_oriented_sphere(q, r, s, t, p)* is returned if q, r, s, and t are non coplanar.

Refines

AdaptableFunctor (with five arguments)

See Also

CGAL::side_of_oriented_sphere . . . . page 1119

11.16 Dimension handling tools

CGAL::Ambient_dimension<T, K>

Definition

The class *Ambient_dimension<T, K>* allows to retrieve the dimension of the ambient space of a type *T* in a kernel *K*.

Parameters

The parameter *K* has the default value *Kernel_traits<T>::Kernel>*.

#include <CGAL/Dimension.h>

Constants

static const int	*value;*	The dimension value as a compile-time integral constant. It is implemented as *K::Dimension<T>::type::value*. It exists only when the dimension is a compile-time constant.

Types

Ambient_dimension<T, K>:: type	Either *Dimension_tag<dim>* if the dimension is a compile-time constant of value *dim*, or *Dynamic_dimension_tag* otherwise. It is implemented as *K::Ambient_dimension<T>::type*.

Example

The following retrieves the dimension of a point type.

```
typedef K::Point_2 Point;
int dimension = Ambient_dimension<Point, K>::value;
assert(dimension == 2);
```

See Also

CGAL::Dimension_tag<int dim> . . . . page 1044
CGAL::Dynamic_dimension_tag . . . . page 1045
CGAL::Feature_dimension<T, K> . . . . page 1043

Class

CGAL::Feature_dimension<T, K>

Definition

The class *Feature_dimension<T, K>* allows to retrieve the geometric dimension of a type *T* in a kernel *K*.

Parameters

The parameter *K* has the default value *Kernel_traits<T>::Kernel*.

#include <CGAL/Dimension.h>

Constants

static const int	*value;*	The dimension value as a compile-time integral constant. It is implemented as *K::Feature_dimension<T> ::type::value*. It exists only when the dimension is a compile-time constant.

Types

Feature_dimension<T, K>:: type	Either *Dimension_tag<dim>* if the dimension is a compile-time constant of value *dim*, or *Dynamic_dimension_tag* otherwise. It is implemented as *K::Feature_dimension<T>::type*.

Example

The following retrieves the dimension of a point type.

```
typedef K::Point_2 Point;
int dimension = Feature_dimension<Point, K>::value;
assert(dimension == 0);
```

See Also

CGAL::Dimension_tag<int dim> . . . . page 1044
CGAL::Dynamic_dimension_tag . . . . page 1045
CGAL::Ambient_dimension<T, K> . . . . page 1042

Class

CGAL::Dimension_tag<int dim>

Definition

An object of the class *Dimension_tag<int dim>* is an empty object which can be used for dispatching functions based on the dimension of an object, as provided by the *dim* parameter. It is useful in cases where it is not more practical to pass the dimension as a template parameter directly.

#include <CGAL/Dimension.h>

Constants

static const int	*value;*	The value of the *dim* parameter.

Example

The following code declares two functions constructing two points at the origin, either in 2D or in 3D.

```
Point_2<K> get_origin(Dimension_tag<2>) { return Point_2<K>(ORIGIN); }
Point_3<K> get_origin(Dimension_tag<3>) { return Point_3<K>(ORIGIN); }

std::cout << get_origin(Dimension_tag<2>())) << std::endl;
```

See Also

CGAL::Ambient_dimension<T, K> . . . . page 1042
CGAL::Feature_dimension<T, K> . . . . page 1043
CGAL::Dynamic_dimension_tag . . . . page 1045

CGAL::Dynamic_dimension_tag

Class

Definition

An object of the class *Dynamic_dimension_tag* is an empty object which can be used for dispatching functions based on the dimension of an object. *Dynamic_dimension_tag* indicates that the dimension is not known at compile-time. *Dimension_tag* is the tag class dealing with compile-time dimensions.

#include <CGAL/Dimension.h>

Example

The following code declares two functions constructing two points at the origin, either in 2D or in 3D.

```
Point_2<K> get_origin(Dimension_tag<2>) { return Point_2<K>(ORIGIN); }
Point_3<K> get_origin(Dimension_tag<3>) { return Point_3<K>(ORIGIN); }
Point_d<K> get_origin(Dynamic_dimension_tag) { return Point_d<K>(ORIGIN); }

std::cout << get_origin(Dynamic_dimension_tag()) << std::endl;
```

See Also

Chapter 12

dD Geometry Kernel

Michael Seel

Contents

12.1 Introduction

This part of the reference manual covers the higher-dimensional kernel. The kernel contains objects of constant size, such as point, vector, direction, line, ray, segment, circle. With each type comes a set of functions which can be applied to an object of this type. You will typically find access functions (e.g. to the coordinates of a point), tests of the position of a point relative to the object, a function returning the bounding box, the length, or the area of an object, and so on. The CGAL kernel further contains basic operations such as affine transformations, detection and computation of intersections, and distance computations. Note that this section partly recapitulates facts already mentioned for the lower-dimensional kernel.

12.1.1 Robustness

The correctness proof of nearly all geometric algorithms presented in theory papers assumes exact computation with real numbers. This leads to a fundamental problem with the implementation of geometric algorithms. Naively, often the exact real arithmetic is replaced by inexact floating-point arithmetic in the implementation. This often leads to acceptable results for many input data. However, even for the implementation of the simplest geometric algorithms this simplification occasionally does not work. Rounding errors introduced by inaccurate arithmetic may lead to inconsistent decisions, causing unexpected failures for some correct input data. There are many approaches to this problem, one of them is to compute exactly (compute so accurate that all decisions made by the algorithm are exact) which is possible in many cases but more expensive than standard floating-point arithmetic. C. M. Hoffmann [Hof89a, Hof89b] illustrates some of the problems arising in the implementation of geometric algorithms and discusses some approaches to solve them. A more recent overview is given in [Sch00]. The exact computation paradigm is discussed by Yap and Dubé [YD95] and Yap [Yap97].

In CGAL you can choose the underlying number types and arithmetic. You can use different types of arithmetic simultaneously and the choice can be easily changed, e.g. for testing. So you can choose between implementations with fast but occasionally inexact arithmetic and implementations guaranteeing exact computation and exact results. Of course you have to pay for the exactness in terms of execution time and storage space. See the dedicated chapter for more details on number types and their capabilities and performance.

12.1.2 Genericity

To increase generic usage of objects and predicates the higher-dimensional kernel makes heavy use of iterator ranges as defined in the STL for modeling tuples. Iterators conceptualize C++ pointers.

For an iterator range *[first,last)* we define *T = tuple [first,last)* as the ordered tuple $(T[0],T[1],\ldots T[d-1])$ where $S[i] = *++^{(i)}\mathit{first}$ (the element obtained by i times forwarding the iterator by operator ++ and then dereferencing it to get the value to which it points). We write *d = size [first,last)* and *S = set [first,last)* to denote the unordered set of elements of the corresponding tuple.

This extends the syntax of random access iterators to input iterators. If we index the tuple as above then we require that $++^{(d+1)}\mathit{first} = \mathit{last}$.

12.2 Kernel Representations

Our object of study is the d-dimensional affine Euclidean space, where d is a parameter of our geometry. Objects in that space are sets of points. A common way to represent the points is the use of Cartesian coordinates, which assumes a reference frame (an origin and d orthogonal axes). In that framework, a point is represented by a

d-tuple $(c_0, c_1, \ldots, c_{d-1})$, and so are vectors in the underlying linear space. Each point is represented uniquely by such Cartesian coordinates.

Another way to represent points is by homogeneous coordinates. In that framework, a point is represented by a $(d+1)$-tuple $(h_0, h_1, \ldots, h_d)$. Via the formulae $c_i = h_i/h_d$, the corresponding point with Cartesian coordinates $(c_0, c_1, \ldots, c_{d-1})$ can be computed. Note that homogeneous coordinates are not unique. For $\lambda \neq 0$, the tuples $(h_0, h_1, \ldots, h_d)$ and $(\lambda \cdot h_0, \lambda \cdot h_1, \ldots, \lambda \cdot h_d)$ represent the same point. For a point with Cartesian coordinates $(c_0, c_1, \ldots, c_{d-1})$ a possible homogeneous representation is $(c_0, c_1, \ldots, c_{d-1}, 1)$. Homogeneous coordinates in fact allow to represent objects in a more general space, the projective space $\mathbb{P}^d$. In CGAL, we do not compute in projective geometry. Rather, we use homogeneous coordinates to avoid division operations, since the additional coordinate can serve as a common denominator.

12.2.1 Genericity through Parameterization

Almost all the kernel objects (and the corresponding functions) are templates with a parameter that allows the user to choose the representation of the kernel objects. A type that is used as an argument for this parameter must fulfill certain requirements on syntax and semantics. The list of requirements defines an abstract kernel concept. In CGAL such a kernel concept is often also called a *representation class* and denoted by *R*. A representation class provides the actual implementations of the kernel objects. For all kernel objects *Kernel_object*, the types *CGAL::Kernel_object*<*R*> and *R::Kernel_object* are identical.

CGAL offers two families of concrete models for the concept representation class, one based on the Cartesian representation of points and one based on the homogeneous representation of points. The interface of the kernel objects is designed such that it works well with both Cartesian and homogeneous representation, for example, points have a constructor with a range of coordinates plus a common denominator (the $d+1$ homogeneous coordinates of the point). The common interfaces parameterized with a representation class allow one to develop code independent of the chosen representation. We said "families" of models, because both families are parameterized too. A user can choose the number type used to represent the coordinates and the linear algebra module used to calculate the result of predicates and constructions.

For reasons that will become evident later, a representation class provides two typenames for number types, namely *R::FT* and *R::RT* and one typename for the linear algebra module *R::LA*. The type *R::FT* must fulfill the requirements on what is called a *field type* in CGAL. This roughly means that *R::FT* is a type for which operations $+$, $-$, $*$ and $/$ are defined with semantics (approximately) corresponding to those of a field in a mathematical sense. Note that, strictly speaking, the built-in type *int* does not fulfill the requirements on a field type, since *int*s correspond to elements of a ring rather than a field, especially operation $/$ is not the inverse of $*$. The requirements on the type *R::RT* are weaker. This type must fulfill the requirements on what is called an *Euclidean ring type* in CGAL. This roughly means that *R::RT* is a type for which operations $+$, $-$, $*$ are defined with semantics (approximately) corresponding to those of a ring in a mathematical sense. A very limited division operation $/$ must be available as well. It must work for exact (i.e., no remainder) integer divisions only. Furthermore, both number types should fulfill CGAL's requirements on a number type.

12.2.2 Cartesian Kernel

With *Cartesian_d*<*FieldNumberType,LinearAlgebra*> you can choose Cartesian representation of coordinates. The type *LinearAlgebra* must me a linear algebra module working on numbers of type *FieldNumberType*. The second parameter defaults to module delivered with the kernel so for short a user can just write *Cartesian_d*< *FieldNumberType*> when not providing her own linear algebra.

When you choose Cartesian representation you have to declare at least the type of the coordinates. A number type used with the *Cartesian_d* representation class should be a *field type* as described above. Both *Carte-*

sian<FieldNumberType>::FT and *Cartesian<FieldNumberType>::RT* are mapped to number type *FieldNumberType*. *Cartesian_d<FieldNumberType,LinearAlgebra>::LA* is mapped to the type *LinearAlgebra*. *Cartesian<FieldNumberType>* uses reference counting internally to save copying costs.

12.2.3 Homogeneous Kernel

As we mentioned before, homogeneous coordinates permit to avoid division operations in numerical computations, since the additional coordinate can serve as a common denominator. Avoiding divisions can be useful for exact geometric computation. With *Homogeneous_d<RingNumberType,LinearAlgebra>* you can choose homogeneous representation of coordinates with the kernel objects. As for Cartesian representation you have to declare at the same time the type used to store the homogeneous coordinates. Since the homogeneous representation allows one to avoid the divisions, the number type associated with a homogeneous representation class must be a model for the weaker concept Euclidean ring type only.

The type *LinearAlgebra* must me a linear algebra module working on numbers of type *RingNumberType*. Again the second parameter defaults to module delivered with the kernel so for short one can just write *Homogeneous_d<RingNumberType>* when replacing the default is no issue.

However, some operations provided by this kernel involve division operations, for example computing squared distances or returning a Cartesian coordinate. To keep the requirements on the number type parameter of *Homogeneous* low, the number type *Quotient<RingNumberType>* is used instead. This number type turns a ring type into a field type. It maintains numbers as quotients, i.e. a numerator and a denominator. Thereby, divisions are circumvented. With *Homogeneous_d<RingNumberType>*, *Homogeneous_d<RingNumberType>::FT* is equal to *Quotient<RingNumberType>* while *Homogeneous_d<RingNumberType>::RT* is equal to *RingNumberType*. *Homogeneous_d<RingNumberType,LinearAlgebra>::LA* is mapped to the type *LinearAlgebra*.

12.2.4 Naming conventions

The use of representation classes not only avoids problems, it also makes all CGAL classes very uniform. They **always** consist of:

1. The *capitalized base name* of the geometric object, such as *Point*, *Segment*, *Triangle*.
2. Followed by *_d*.
3. A *representation class* as parameter, which itself is parameterized with a number type, such as *Cartesian_d<double>* or *Homogeneous_d<leda_integer>*.

12.2.5 Kernel as a Traits Class

Algorithms and data structures in the basic library of CGAL are parameterized by a traits class that subsumes the objects on which the algorithm or data structure operates as well as the operations to do so. For most of the algorithms and data structures in the basic library you can use a kernel as a traits class. For some algorithms you even do not have to specify the kernel; it is detected automatically using the types of the geometric objects passed to the algorithm. In some other cases, the algorithms or data structures needs more than is provided by a kernel. In these cases, a kernel can not be used as a traits class.

12.2.6 Choosing a Kernel

If you start with integral Cartesian coordinates, many geometric computations will involve integral numerical values only. Especially, this is true for geometric computations that evaluate only predicates, which are tantamount to determinant computations. Examples are triangulation of point sets and convex hull computation.

The dimension d of our affine space determines the dimension of the matrix computations in the mathematical evaluation of predicates. As rounding errors accumulate fast the homogeneous representation used with multi-precision integers is the kernel of choice for well-behaved algorithms. Note, that unless you use an arbitrary precision integer type, incorrect results might arise due to overflow.

If new points are to be constructed, for example the intersection point of two lines, computation of Cartesian coordinates usually involves divisions, so you need to use a field type with Cartesian representation or have to switch to homogeneous representation. *double* is a possible, but imprecise field type. You can also put any ring type into *Quotient* to get a field type and put it into *Cartesian*, but you better put the ring type into *Homogeneous*. *leda_rational* and *leda_real* are valid field types, too.

Still other people will prefer the built-in type `double`, because they need speed and can live with approximate results, or even algorithms that, from time to time, crash or compute incorrect results due to accumulated rounding errors.

12.2.7 Inclusion of Header Files

You need just to include a representation class to obtain the geometric objects of the kernel that you would like to use with the representation class, i.e., *CGAL/Cartesian_d.h* or *CGAL/Homogeneous_d.h*

12.3 Kernel Geometry

12.3.1 Points and Vectors

In CGAL, we strictly distinguish between points, vectors and directions. A *point* is a point in the Euclidean space $\mathbb{E}^d$, a *vector* is the difference of two points p_2, p_1 and denotes the direction and the distance from p_1 to p_2 in the vector space $\mathbb{R}^d$, and a *direction* is a vector where we forget about its length. They are different mathematical concepts. For example, they behave different under affine transformations and an addition of two points is meaningless in affine geometry. By putting them in different classes we not only get cleaner code, but also type checking by the compiler which avoids ambiguous expressions. Hence, it pays twice to make this distinction.

CGAL defines a symbolic constant *ORIGIN* of type *Origin* which denotes the point at the origin. This constant is used in the conversion between points and vectors. Subtracting it from a point p results in the locus vector of p.

```
double coord[] = {1.0, 1.0, 1.0, 1.0};
Point_d< Cartesian_d<double> > p(4,coord,coord+4), q(4);
Vector_d< Cartesian_d<double> > v(4);
v = p - ORIGIN;
q = ORIGIN + v;
assert( p == q );
```

In order to obtain the point corresponding to a vector v you simply have to add v to *ORIGIN*. If you want to determine the point q in the middle between two points p_1 and p_2, you can write[1]

```
q = p_1 + (p_2 - p_1) / 2.0;
```

Note that these constructions do not involve any performance overhead for the conversion with the currently available representation classes.

12.3.2 Kernel Objects

Besides points (*Point_d*<*R*>), vectors (*Vector_d*<*R*>), and directions (*Direction_d*<*R*>), CGAL provides lines, rays, segments, hyperplanes, and spheres.

Lines (*Line_d*<*R*>) in CGAL are oriented. A ray (*Ray_d*<*R*>) is a semi-infinite interval on a line, and this line is oriented from the finite endpoint of this interval towards any other point in this interval. A segment (*Segment_d*< *R*>) is a bounded interval on a directed line, and the endpoints are ordered so that they induce the same direction as that of the line.

Hyperplanes are affine subspaces of dimension $d-1$ in $\mathbb{E}^d$, passing through d points. Hyperplanes are oriented and partition space into a positive side and a negative side. In CGAL, there are no special classes for halfspaces. Halfspaces are supposed to be represented by oriented hyperplanes. All kernel objects are equality comparable via *operator*== and *operator!*=. For those oriented objects whose orientation can be reversed (segments, lines, hyperplanes, spheres) there is also a global function *weak_equality* that allows to test for point set equality disregarding the orientation.

12.3.3 Orientation and Relative Position

Geometric objects in CGAL have member functions that test the position of a point relative to the object. Full dimensional objects and their boundaries are represented by the same type, e.g. halfspaces and hyperplanes are not distinguished, neither are balls and spheres. Such objects split the ambient space into two full-dimensional parts, a bounded part and an unbounded part (e.g. spheres), or two unbounded parts (e.g. hyperplanes). By default these objects are oriented, i.e., one of the resulting parts is called the positive side, the other one is called the negative side. Both of these may be unbounded.

For these objects there is a function *oriented_side()* that determines whether a test point is on the positive side, the negative side, or on the oriented boundary. These function returns a value of type *Oriented_side*.

Those objects that split the space in a bounded and an unbounded part, have a member function *bounded_side()* with return type *Bounded_side*.

If an object is lower dimensional, e.g. a segment in d-dimensional space, there is only a test whether a point belongs to the object or not. This member function, which takes a point as an argument and returns a Boolean value, is called *has_on()*

[1] you might call *midpoint(p_1,p_2)* instead

12.4 Predicates and Constructions

12.4.1 Predicates

Predicates are at the heart of a geometry kernel. They are basic units for the composition of geometric algorithms and encapsulate decisions. Hence their correctness is crucial for the control flow and hence for the correctness of an implementation of a geometric algorithm. CGAL uses the term predicate in a generalized sense. Not only components returning a Boolean value are called predicates but also components returning an enumeration type like a *Comparison_result* or an *Orientation*. We say components, because predicates are implemented both as functions and function objects (also called functors and provided by a kernel class).

CGAL provides predicates for the orientation of point sets (*orientation*), for comparing points according to some given order, especially for comparing Cartesian coordinates (e.g. *lexicographically_xy_smaller*), in-sphere tests, and predicates to compare distances.

12.4.2 Constructions

Functions and function objects that generate objects that are neither of type *bool* nor enum types are called constructions. Constructions involve computation of new numerical values and may be imprecise due to rounding errors unless a kernel with an exact number type is used.

Affine transformations (*Aff_transformation_d<R>*) allow to generate new object instances under arbitrary affine transformations. These transformations include translations, rotations (within planes) and scaling. Most of the geometric objects in a kernel have a member function *transform(Aff_transformation t)* which applies the transformation to the object instance.

CGAL also provides a set of functions that detect or compute the intersection between objects and functions to calculate their squared distance. Moreover, some member functions of kernel objects are constructions.

So there are routines that compute the square of the Euclidean distance, but no routines that compute the distance itself. Why? First of all, the two values can be derived from each other quite easily (by taking the square root or taking the square). So, supplying only the one and not the other is only a minor inconvenience for the user. Second, often either value can be used. This is for example the case when (squared) distances are compared. Third, the library wants to stimulate the use of the squared distance instead of the distance. The squared distance can be computed in more cases and the computation is cheaper. We do this by not providing the perhaps more natural routine, The problem of a distance routine is that it needs the *sqrt* operation. This has two drawbacks:

- The *sqrt* operation can be costly. Even if it is not very costly for a specific number type and platform, avoiding it is always cheaper.
- There are number types on which no *sqrt* operation is defined, especially integer types and rationals.

12.4.3 Intersection and Polymorphic Return Values

Intersections on kernel objects currently cover only those objects that are part of flats (*Segment_d<R>*, *Ray_d<R>*, *Line_c<R>*, and *Hyperplane_d<R>*). For any pair of objects $o1$, $o2$ of these types the operation *intersection(o1,o2)* returns a polymorphic object that wraps the result of the intersection operation.

The class *Object* provides the polymorphic abstraction. An object *obj* of type *Object* can represent an arbitrary class. The only operations it provides is to make copies and assignments, so that you can put them in lists or arrays. Note that *Object* is NOT a common base class for the elementary classes. Therefore, there is no automatic conversion from these classes to *Object* Rather this is done with the global function *make_object()*. This encapsulation mechanism requires the use of *object_cast* to unwrap the encapsulated class.

Example

In the following example, the object type is used as a return value for the intersection computation, as there are possibly different return values.

```
typedef Point_d< Cartesian_d<double> >    Point;
typedef Segment_d< Cartesian_d<double> >  Segment;
Segment s1, s2;
std::cin >> s1 >> s2;
Object obj = intersection(s1, s2);
if (const Point *p = object_cast<Point>(&obj) ) {
  /* do something with *p */
} else if (const Segment *s = object_cast&lt;Segment&gt;(&obj) ) {
  /* do something with *s */
}
/* there was no intersection */
```

12.4.4 Constructive Predicates

For testing where a point p lies with respect to a hyperplane defined by an array P of points $p_1, \dots, p_d$, one may be tempted to construct the hyperplane *Hyperplane_d<R>(d,P,P+d)* and use the method *oriented_side(p)*. This may pay off if many tests with respect to the plane are made. Nevertheless, unless the number type is exact, the constructed plane is only approximated, and round-off errors may lead *oriented_side(p)* to return an orientation which is different from the orientation of $p_1, \dots, p_d, p$.

In CGAL, we provide predicates in which such geometric decisions are made directly with a reference to the input points in P without an intermediary object like a plane. For the above test, the recommended way to get the result is to use *orientation(P',P'+d)*, where P' is an array containing the points $p_1, \dots, p_d, p$.

For exact number types like *leda_real*, the situation is different. If several tests are to be made with the same plane, it pays off to construct the plane and to use *oriented_side(p)*.

12.5 Design and Implementation History

This higher-dimensional kernel is the result of a long evolving development. A first version of the kernel was offered as a LEDA extension package ddgeo by Kurt Mehlhorn and Michael Seel. The original design was driven by the realization of a d-dimensional convex hull data type developed at the Max-Planck Institut für Informatik.

The code base was discussed and reviewed within the CGAL kernel group (of the low-dimensional kernel). This led to the identification of the concept interfaces and in parallel to adaptations according to the evolution of the low-dimensional kernel. The kernel was revised based on suggestions by Hervé Brönnimann, Michael Hoffmann, and Stefan Schirra.

12.5.1 Acknowledgments

This work was supported by ESPRIT IV Long Term Research Projects No. 21957 (CGAL) and No. 28155 (GALIA).

dD Geometry Kernel Reference Manual

Michael Seel

Concepts

Classes

Predicates and Construction on Points

Predicates and Construction on Vectors

Intersection operations

12.6 Linear Algebra Concepts and Classes

LinearAlgebraTraits_d

Concept

Definition

The data type *LinearAlgebraTraits_d* encapsulates two classes *Matrix*, *Vector* and many functions of basic linear algebra. An instance of data type *Matrix* is a matrix of variables of type *NT*. Accordingly, *Vector* implements vectors of variables of type *NT*. Most functions of linear algebra are *checkable*, i.e., the programs can be asked for a proof that their output is correct. For example, if the linear system solver declares a linear system $Ax = b$ unsolvable it also returns a vector c such that $c^T A = 0$ and $c^T b \neq 0$.

Types

LinearAlgebraTraits_d:: NT — the number type of the components.

LinearAlgebraTraits_d:: Vector — the vector type.

LinearAlgebraTraits_d:: Matrix — the matrix type.

Operations

static Matrix *LA.transpose(Matrix M)*

returns M^T (a *M.column_dimension()* $\times$ *M.column_dimension()* - matrix).

static bool *LA.inverse(Matrix M, Matrix& I, NT& D, Vector& c)*

determines whether M has an inverse. It also computes either the inverse as $(1/D) \cdot I$ or when no inverse exists, a vector c such that $c^T \cdot M = 0$.
Precondition: M is square.

static Matrix *LA.inverse(Matrix M, NT& D)*

returns the inverse matrix of M. More precisely, $1/D$ times the matrix returned is the inverse of M.

Precondition: *determinant(M) != 0*.
Precondition: M is square.

static NT *LA.determinant(Matrix M, Matrix& L, Matrix& U, std::vector<int>& q, Vector& c)*

returns the determinant D of M and sufficient information to verify that the value of the determinant is correct. If the determinant is zero then c is a vector such that $c^T \cdot M = 0$. If the determinant is non-zero then L and U are lower and upper diagonal matrices respectively and q encodes a permutation matrix Q with $Q(i,j) = 1$ iff $i = q(j)$ such that $L \cdot M \cdot Q = U$, $L(0,0) = 1$, $L(i,i) = U(i-1,i-1)$ for all i, $1 \leq i < n$, and $D = s \cdot U(n-1,n-1)$ where s is the determinant of Q.
Precondition: M is square.

static bool *LA.verify_determinant(Matrix M,*
NT D,
Matrix& L,
Matrix& U,
std::vector<int> q,
Vector& c)

verifies the conditions stated above.

static NT *LA.determinant(Matrix M)*

returns the determinant of M.
Precondition: M is square.

static int *LA.sign_of_determinant(Matrix M)*

returns the sign of the determinant of M.
Precondition: M is square.

static bool *LA.linear_solver(Matrix M,*
Vector b,
Vector& x,
NT& D,
Matrix& spanning_vectors,
Vector& c)

determines the complete solution space of the linear system $M \cdot x = b$. If the system is unsolvable then $c^T \cdot M = 0$ and $c^T \cdot b \neq 0$. If the system is solvable then $(1/D)x$ is a solution, and the columns of *spanning_vectors* are a maximal set of linearly independent solutions to the corresponding homogeneous system.
Precondition: *M.row_dimension() = b.dimension()*.

static bool *LA.linear_solver(Matrix M, Vector b, Vector& x, NT& D, Vector& c)*

determines whether the linear system $M \cdot x = b$ is solvable. If yes, then $(1/D)x$ is a solution, if not then $c^T \cdot M = 0$ and $c^T \cdot b \neq 0$.
Precondition: *M.row_dimension() = b.dimension()*.

static bool *LA.linear_solver(Matrix M, Vector b, Vector& x, NT& D)*

as above, but without the witness c
Precondition: *M.row_dimension() = b.dimension()*.

static bool *LA.is_solvable(Matrix M, Vector b)*

determines whether the system $M \cdot x = b$ is solvable

Precondition: *M.row_dimension() = b.dimension()*.

static bool *LA.homogeneous_linear_solver(Matrix M, Vector& x)*

determines whether the homogeneous linear system $M \cdot x = 0$ has a non - trivial solution. If yes, then x is such a solution.

static int *LA.homogeneous_linear_solver(Matrix M, Matrix& spanning_vecs)*

determines the solution space of the homogeneous linear system $M \cdot x = 0$. It returns the dimension of the solution space. Moreover the columns of *spanning_vecs* span the solution space.

static int *LA.independent_columns(Matrix M, std::vector<int>& columns)*

returns the indices of a maximal subset of independent columns of M.

static int *LA.rank(Matrix M)*

returns the rank of matrix M

Has Models

CGAL::Linear_algebraHd<RT> .... page 1068
CGAL::Linear_algebraCd<FT> .... page 1067

Concept

Vector

Definition

An instance of data type *Vector* is a vector of variables of number type *NT*. Together with the type *Matrix* it realizes the basic operations of linear algebra.

Types

Vector:: NT	the ring type of the components.
Vector:: iterator	the iterator type for accessing components.
Vector:: const_iterator	the const iterator type for accessing components.

Creation

Vector v;	creates an instance *v* of type *Vector*.
Vector v(int d);	creates an instance *v* of type *Vector*. *v* is initialized to a vector of dimension *d*.
Vector v(int d, NT x);	creates an instance *v* of type *Vector*. *v* is initialized to a vector of dimension *d* with entries *x*.

template <class Forward_iterator>
Vector v(Forward_iterator first, Forward_iterator last);

creates an instance *v* of type *Vector*; *v* is initialized to the vector with entries *set [first,last)*.
Requirement: *Forward_iterator* has value type *NT*.

Operations

int	*v.dimension()*	returns the dimension of *v*.
bool	*v.is_zero()*	returns true iff *v* is the zero vector.
NT&	*v*[*int i*]	returns the *i*-th component of *v*. *Precondition*: $0 \leq i \leq v.dimension() - 1$.
iterator	*v.begin()*	iterator to the first component.
iterator	*v.end()*	iterator beyond the last component.

The same operations *begin()*, *end()* exist for *const_iterator*.

Vector	$v + v1$	Addition. *Precondition*: *v.dimension() == v1.dimension()*.
Vector	$v - v1$	Subtraction. *Precondition*: *v.dimension() = v1.dimension()*.
NT	$v * v1$	Inner Product. *Precondition*: *v.dimension() = v1.dimension()*.
Vector	$-v$	Negation.
Vector&	$v += v1$	Addition plus assignment. *Precondition*: *v.dimension() == v1.dimension()*.
Vector&	$v -= v1$	Subtraction plus assignment. *Precondition*: *v.dimension() == v1.dimension()*.
Vector&	$v *= NT\ s$	Scalar multiplication plus assignment.
Vector&	$v /= NT\ s$	Scalar division plus assignment.
Vector	$NT\ r * v$	Component-wise multiplication with number *r*.
Vector	$v * NT\ r$	Component-wise multiplication with number *r*.

Matrix

Definition

An instance of data type *Matrix* is a matrix of variables of number type *NT*. The types *Matrix* and *Vector* together realize many functions of basic linear algebra.

Types

Matrix:: NT	the ring type of the components.
Matrix:: iterator	bidirectional iterator for accessing all components row-wise.
Matrix:: row_iterator	random access iterator for accessing row entries.
Matrix:: column_iterator	random access iterator for accessing column entries.

There also constant versions of the above iterators: *const_iterator*, *row_const_iterator*, and *column_const_iterator*.

Matrix:: Identity	a tag class for identity initialization
Matrix:: Vector	the vector type used.

Creation

Matrix M;	creates an instance M of type *Matrix*.
Matrix M(int n);	creates an instance M of type *Matrix* of dimension $n \times n$ initialized to the zero matrix.
Matrix M(int m, int n);	creates an instance M of type *Matrix* of dimension $m \times n$ initialized to the zero matrix.
Matrix M(std::pair<int,int> p);	creates an instance M of type *Matrix* of dimension *p.first*$\times$*p.second* initialized to the zero matrix.

Matrix M(int n, Identity, NT x = NT(1));

creates an instance M of type *Matrix* of dimension $n \times n$ initialized to the identity matrix (times x).

Matrix M(int m, int n, NT x);	creates an instance M of type *Matrix* of dimension $m \times n$ initialized to the matrix with x entries.

template <class Forward_iterator>
Matrix M(Forward_iterator first, Forward_iterator last);

creates an instance M of type *Matrix*. Let S be the ordered set of n column-vectors of common dimension m as given by the iterator range *[first,last)*. M is initialized to an $m \times n$ matrix with the columns as specified by S.
Precondition: *Forward_iterator* has a value type V from which we require to provide a iterator type *V::const_iterator*, to have *V::value_type* $== NT$.
Note that *Vector* or *std::vector<NT>* fulfill these requirements.

Matrix M(std::vector< Vector > A);

creates an instance M of type *Matrix*. Let A be an array of n column-vectors of common dimension m. M is initialized to an $m \times n$ matrix with the columns as specified by A.

Operations

int	*M.row_dimension()*	returns n, the number of rows of M.
int	*M.column_dimension()*	returns m, the number of columns of M.
std::pair<int,int>	*M.dimension()*	returns (m,n), the dimension pair of M.
Vector	*M.row(int i)*	returns the i-th row of M (an m - vector). *Precondition*: $0 \leq i \leq m-1$.
Vector	*M.column(int i)*	returns the i-th column of M (an n - vector). *Precondition*: $0 \leq i \leq n-1$.
NT&	*M(int i, int j)*	returns $M_{i,j}$. *Precondition*: $0 \leq i \leq m-1$ and $0 \leq j \leq n-1$.
void	*M.swap_rows(int i, int j)*	swaps rows i and j. *Precondition*: $0 \leq i \leq m-1$ and $0 \leq j \leq m-1$.
void	*M.swap_columns(int i, int j)*	swaps columns i and j. *Precondition*: $0 \leq i \leq n-1$ and $0 \leq j \leq n-1$.
row_iterator	*M.row_begin(int i)*	an iterator pointing to the first entry of the ith row. *Precondition*: $0 \leq i \leq m-1$.
row_iterator	*M.row_end(int i)*	an iterator pointing beyond the last entry of the ith row. *Precondition*: $0 \leq i \leq m-1$.

column_iterator	*M.column_begin(int i)*	an iterator pointing to the first entry of the ith column. *Precondition*: $0 \leq i \leq n-1$.
column_iterator	*M.column_end(int i)*	an iterator pointing beyond the last entry of the ith column. *Precondition*: $0 \leq i \leq n-1$.
iterator	*M.begin()*	an iterator pointing to the first entry of M.
terator	*M.end()*	an iterator pointing beyond the last entry of M.

The same operations exist for *row_const_iterator*, *column_const_iterator* and *const_iterator*.

bool	$M == M1$	Test for equality.
bool	$M \,!= M1$	Test for inequality.

Arithmetic Operators

Matrix	$M + M1$	Addition. *Precondition*: *M.row_dimension() == M1.row_dimension()* and *M.column_dimension() == M1.column_dimension()*.
Matrix	$M - M1$	Subtraction. *Precondition*: *M.row_dimension() == M1.row_dimension()* and *M.column_dimension() == M1.column_dimension()*.
Matrix	$-M$	Negation.
Matrix	$M * M1$	Multiplication. *Precondition*: *M.column_dimension() = M1.row_dimension()*.
Vector	$M *$ *Vector vec*	Multiplication with vector. *Precondition*: *M.column_dimension() = vec.dimension()*.
Matrix	*NT* $x * M$	Multiplication of every entry with x.
Matrix	$M *$ *NT* x	Multiplication of every entry with x.

CGAL::Linear_algebraCd<FT>

Definition

The class *Linear_algebraCd<FT>* serves as the default traits class for the LA parameter of *CGAL::Cartesian_d<FT,LA>*. It implements linear algebra for field number types *FT*.

#include <CGAL/Linear_algebraCd.h>

Is Model for the Concepts

LinearAlgebraTraits_d . . . . page 1059

Requirements

FT must be a field number type.

Operations

Fits all operation requirements of the concept.

Class

CGAL::Linear_algebraHd<RT>

Definition

The class *Linear_algebraHd<RT>* serves as the default traits class for the LA parameter of *CGAL::Homogeneous_d<RT,LA>*. It implements linear algebra for Euclidean ring number types *RT*.

#include <CGAL/Linear_algebraHd.h>

Is Model for the Concepts

LinearAlgebraTraits_d . . . . page 1059

Requirements

To make a ring number type *RT* work with this class it has to provide a division *operator/* with remainder.

Operations

Fits all operation requirements of the concept.

12.7 Kernels

CGAL::Cartesian_d<FieldNumberType>

#include <CGAL/Cartesian_d.h>

Definition

A model for *Kernel_d* that uses Cartesian coordinates to represent the geometric objects. In order for *Cartesian_d<FieldNumberType>* to model Euclidean geometry in E^d , for some mathematical field E (*e.g.*, the rationals $\mathbb{Q}$ or the reals $\mathbb{R}$), the template parameter *FieldNumberType* must model the mathematical field E. That is, the field operations on this number type must compute the mathematically correct results. If the number type provided as a model for *FieldNumberType* is only an approximation of a field (such as the built-in type *double*), then the geometry provided by the kernel is only an approximation of Euclidean geometry.

Is Model for the Concepts

Kernel_d . . . . page 1121

See Also

CGAL::Homogeneous_d<RingNumberType> . . . . page **??**

Class

Class

CGAL::Homogeneous<RingNumberType>

#include <CGAL/Homogeneous_d.h>

Definition

A model for a *Kernel_d* using homogeneous coordinates to represent the geometric objects. In order for *Homogeneous<RingNumberType>* to model Euclidean geometry in E^d, for some mathematical ring E (*e.g.*, the integers $\mathbb{Z}$ or the rationals $\mathbb{Q}$), the template parameter *RT* must model the mathematical ring E. That is, the ring operations on this number type must compute the mathematically correct results. If the number type provided as a model for *RingNumberType* is only an approximation of a ring (such as the built-in type *double*), then the geometry provided by the kernel is only an approximation of Euclidean geometry.

Is Model for the Concepts

Kernel_d . . . . page 1121

See Also

CGAL::Cartesian_d<FieldumberType> . . . . page **??**

12.8 Kernel Objects

CGAL::Point_d<Kernel>

Definition

An instance of data type *Point_d<Kernel>* is a point of Euclidean space in dimension d. A point $p = (p_0, \ldots, p_{d-1})$ in d-dimensional space can be represented by homogeneous coordinates $(h_0, h_1, \ldots, h_d)$ of number type RT such that $p_i = h_i/h_d$, which is of type FT. The homogenizing coordinate h_d is positive.

We call p_i, $0 \leq i < d$ the i-th Cartesian coordinate and h_i, $0 \leq i \leq d$, the i-th homogeneous coordinate. We call d the dimension of the point.

Types

Point_d<Kernel>:: LA the linear algebra layer.

Point_d<Kernel>:: Cartesian_const_iterator

a read-only iterator for the Cartesian coordinates.

Point_d<Kernel>:: Homogeneous_const_iterator

a read-only iterator for the homogeneous coordinates.

Creation

Point_d<Kernel> p; introduces a variable p of type *Point_d<Kernel>*.

Point_d<Kernel> p(int d, Origin);

introduces a variable p of type *Point_d<Kernel>* in d-dimensional space, initialized to the origin.

template <class InputIterator>
Point_d<Kernel> p(int d, InputIterator first, InputIterator last);

introduces a variable p of type *Point_d<Kernel>* in dimension d. If *size [first,last) == d* this creates a point with Cartesian coordinates *set [first,last)*. If *size [first,last) == d+1* the range specifies the homogeneous coordinates $H = set\,[first, last) = (\pm h_0, \pm h_1, \ldots, \pm h_d)$ where the sign chosen is the sign of h_d.
Precondition: d is nonnegative, *[first,last)* has d or *d+1* elements where the last has to be non-zero.
Requirement: The value type of *InputIterator* is RT.

template <class InputIterator>
Point_d<Kernel> p(int d, InputIterator first, InputIterator last, RT D);

introduces a variable *p* of type *Point_d<Kernel>* in dimension *d* initialized to the point with homogeneous coordinates as defined by *H = set [first,last)* and *D*: $(\pm H[0], \pm H[1], \ldots, \pm H[d-1], \pm D)$. The sign chosen is the sign of *D*.
Precondition: *D* is non-zero, the iterator range defines a *d*-tuple of *RT*.
Requirement: The value type of *InputIterator* is *RT*.

Point_d<Kernel> p(RT x, RT y, RT w = 1);

introduces a variable *p* of type *Point_d<Kernel>* in 2-dimensional space.
Precondition: $w \neq 0$.

Point_d<Kernel> p(RT x, RT y, RT z, RT w);

introduces a variable *p* of type *Point_d<Kernel>* in 3-dimensional space.
Precondition: $w \neq 0$.

Operations

int	*p.dimension()*	returns the dimension of *p*.
FT	*p.cartesian(int i)*	returns the *i*-th Cartesian coordinate of *p*. *Precondition*: $0 \leq i < d$.
FT	*p*[*int i*]	returns the *i*-th Cartesian coordinate of *p*. *Precondition*: $0 \leq i < d$.
RT	*p.homogeneous(int i)*	returns the *i*-th homogeneous coordinate of *p*. *Precondition*: $0 \leq i \leq d$.
Cartesian_const_iterator	*p.cartesian_begin()*	returns an iterator pointing to the zeroth Cartesian coordinate p_0 of *p*.
Cartesian_const_iterator	*p.cartesian_end()*	returns an iterator pointing beyond the last Cartesian coordinate of *p*.
Homogeneous_const_iterator	*p.homogeneous_begin()*	returns an iterator pointing to the zeroth homogeneous coordinate h_0 of *p*.

Homogeneous_const_iterator

p.homogeneous_end()

returns an iterator pointing beyond the last homogeneous coordinate of *p*.

Point_d<Kernel> *p.transform(Aff_transformation_d<Kernel> t)*

returns $t(p)$.

Arithmetic Operators, Tests and IO

Vector_d<Kernel> $p -$ *Origin o* returns the vector $p - O$.

Vector_d<Kernel> $p - q$ returns $p - q$.
Precondition: *p.dimension()* == *q.dimension()*.

Point_d<Kernel> $p +$ *Vector_d<Kernel> v*

returns $p + v$.
Precondition: *p.dimension()* == *v.dimension()*.

Point_d<Kernel> $p -$ *Vector_d<Kernel> v*

returns $p - v$.
Precondition: *p.dimension()* == *v.dimension()*.

Point_d<Kernel>& $p +=$ *Vector_d<Kernel> v*

adds *v* to *p*.

Precondition: *p.dimension()* == *v.dimension()*.

Point_d<Kernel>& $p -=$ *Vector_d<Kernel> v*

subtracts *v* from *p*.

Precondition: *p.dimension()* == *v.dimension()*.

bool $p ==$ *Origin* returns true if *p* is the origin.

Downward compatibility

We provide operations of the lower dimensional interface *x()*, *y()*, *z()*, *hx()*, *hy()*, *hz()*, *hw()*.

Implementation

Points are implemented by arrays of *RT* items. All operations like creation, initialization, tests, point - vector arithmetic, input and output on a point p take time $O(p.dimension())$. *dimension()*, coordinate access and conversions take constant time. The space requirement for points is $O(p.dimension())$.

CGAL::Vector_d<Kernel>

Class

Definition

An instance of data type *Vector_d<Kernel>* is a vector of Euclidean space in dimension d. A vector $r = (r_0, \ldots, r_{d-1})$ can be represented in homogeneous coordinates $(h_0, \ldots, h_d)$ of number type RT, such that $r_i = h_i / h_d$ which is of type FT. We call the r_i's the Cartesian coordinates of the vector. The homogenizing coordinate h_d is positive.

This data type is meant for use in computational geometry. It realizes free vectors as opposed to position vectors (type *Point_d*). The main difference between position vectors and free vectors is their behavior under affine transformations, e.g., free vectors are invariant under translations.

Types

Vector_d<Kernel>:: LA — the linear algebra layer.

Vector_d<Kernel>:: Cartesian_const_iterator

a read-only iterator for the Cartesian coordinates.

Vector_d<Kernel>:: Homogeneous_const_iterator

a read-only iterator for the homogeneous coordinates.

Vector_d<Kernel>:: Base_vector

construction tag.

Creation

Vector_d<Kernel> v; — introduces a variable v of type *Vector_d<Kernel>*.

Vector_d<Kernel> v(int d, Null_vector);

introduces the zero vector v of type *Vector_d<Kernel>* in d-dimensional space. For the creation flag *CGAL::NULL_VECTOR* can be used.

template <class InputIterator>
Vector_d<Kernel> v(int d, InputIterator first, InputIterator last);

introduces a variable v of type *Vector_d<Kernel>* in dimension d. If *size [first,last) == d* this creates a vector with Cartesian coordinates *set [first,last)*. If *size [first,last) == p+1* the range specifies the homogeneous coordinates $H = set\ [first, last) = (\pm h_0, \pm h_1, \ldots, \pm h_d)$ where the sign chosen is the sign of h_d.
Precondition: d is nonnegative, *[first,last)* has d or d*+1* elements where the last has to be non-zero.
Requirement: The value type of *InputIterator* is RT.

template <class InputIterator>
Vector_d<Kernel> v(int d, InputIterator first, InputIterator last, RT D);

introduces a variable v of type *Vector_d<Kernel>* in dimension d initialized to the vector with homogeneous coordinates as defined by H = *set [first,last)* and D: $(\pm H[0], \pm H[1], \ldots, \pm H[d-1], \pm D)$. The sign chosen is the sign of D.
Precondition: D is non-zero, the iterator range defines a d-tuple of RT.
Requirement: The value type of *InputIterator* is RT.

Vector_d<Kernel> v(int d, Base_vector, int i);

returns a variable v of type *Vector_d<Kernel>* initialized to the i-th base vector of dimension d.
Precondition: $0 \leq i < d$.

Vector_d<Kernel> v(RT x, RT y, RT w = 1);

introduces a variable v of type *Vector_d<Kernel>* in 2-dimensional space.
Precondition: $w \neq 0$.

Vector_d<Kernel> v(RT x, RT y, RT z, RT w);

introduces a variable v of type *Vector_d<Kernel>* in 3-dimensional space.
Precondition: $w \neq 0$.

Operations

int	*v.dimension()*	returns the dimension of v.
FT	*v.cartesian(int i)*	returns the i-th Cartesian coordinate of v. *Precondition*: $0 \leq i < d$.
FT	*v*[*int i*]	returns the i-th Cartesian coordinate of v. *Precondition*: $0 \leq i < d$.
RT	*v.homogeneous(int i)*	returns the i-th homogeneous coordinate of v. *Precondition*: $0 \leq i \leq d$.

FT	*v.squared_length()*	returns the square of the length of v.
Cartesian_const_iterator	*v.cartesian_begin()*	returns an iterator pointing to the zeroth Cartesian coordinate of v.
Cartesian_const_iterator	*v.cartesian_end()*	returns an iterator pointing beyond the last Cartesian coordinate of v.
Homogeneous_const_iterator	*v.homogeneous_begin()*	returns an iterator pointing to the zeroth homogeneous coordinate of v.
Homogeneous_const_iterator	*v.homogeneous_end()*	returns an iterator pointing beyond the last homogeneous coordinate of v.
Direction_d<Kernel>	*v.direction()*	returns the direction of v.
Vector_d<Kernel>	*v.transform(Aff_transformation_d<Kernel> t)*	returns $t(v)$.

Arithmetic Operators, Tests and IO

Vector_d<Kernel>&	$v *= RT\ n$	multiplies all Cartesian coordinates by n.
Vector_d<Kernel>&	$v *= FT\ r$	multiplies all Cartesian coordinates by r.
Vector_d<Kernel>	$v\,/RT\ n$	returns the vector with Cartesian coordinates $v_i/n, 0 \leq i < d$.
Vector_d<Kernel>	$v\,/FT\ r$	returns the vector with Cartesian coordinates $v_i/r, 0 \leq i < d$.
Vector_d<Kernel>&	$v\,/= RT\ n$	divides all Cartesian coordinates by n.
Vector_d<Kernel>&	$v\,/= FT\ r$	divides all Cartesian coordinates by r.
FT	$v * w$	inner product, i.e., $\sum_{0 \leq i < d} v_i w_i$, where v_i and w_i are the Cartesian coordinates of v and w respectively.
Vector_d<Kernel>	$v + w$	returns the vector with Cartesian coordinates $v_i + w_i, 0 \leq i < d$.
Vector_d<Kernel>&	$v += w$	addition plus assignment.

Vector_d<Kernel>	$v - w$	returns the vector with Cartesian coordinates $v_i - w_i, 0 \leq i < d$.
Vector_d<Kernel>&	$v -= w$	subtraction plus assignment.
Vector_d<Kernel>	$-v$	returns the vector in opposite direction.
bool	*v.is_zero()*	returns true if v is the zero vector.

Downward compatibility

We provide all operations of the lower dimensional interface *x()*, *y()*, *z()*, *hx()*, *hy()*, *hz()*, *hw()*.

Vector_d<Kernel>	*RT* $n * v$	returns the vector with Cartesian coordinates nv_i.
Vector_d<Kernel>	*FT* $r * v$	returns the vector with Cartesian coordinates $rv_i, 0 \leq i < d$.

Implementation

Vectors are implemented by arrays of variables of type RT. All operations like creation, initialization, tests, vector arithmetic, input and output on a vector v take time $O(v.dimension())$. coordinate access, *dimension()* and conversions take constant time. The space requirement of a vector is $O(v.dimension())$.

CGAL::Direction_d<Kernel>

Definition

A *Direction_d* is a vector in the d-dimensional vector space where we forget about its length. We represent directions in d-dimensional space as a tuple $(h_0, \ldots, h_d)$ of variables of type *RT* which we call the homogeneous coordinates of the direction. The coordinate h_d must be positive. The Cartesian coordinates of a direction are $c_i = h_i/h_d$ for $0 \leq i < d$, which are of type *FT*. Two directions are equal if their Cartesian coordinates are positive multiples of each other. Directions are in one-to-one correspondence to points on the unit sphere.

Types

Direction_d<Kernel>:: LA the linear algebra layer.

Direction_d<Kernel>:: Delta_const_iterator

a read-only iterator for the deltas of *dir*.

Direction_d<Kernel>:: Base_direction

construction tag.

Creation

Direction_d<Kernel> dir; introduces a variable *dir* of type *Direction_d<Kernel>*.

Direction_d<Kernel> dir(Vector_d<Kernel> v);

introduces a variable *dir* of type *Direction_d<Kernel>* initialized to the direction of v.

template <class InputIterator>
Direction_d<Kernel> dir(int d, InputIterator first, InputIterator last);

introduces a variable *dir* of type *Direction_d<Kernel>* in dimension d with representation tuple *set [first,last)*.
Precondition: d is nonnegative, *[first,last)* has d elements.
Requirement: The value type of *InputIterator* is *RT*.

Direction_d<Kernel> dir(int d, Base_direction, int i);

returns a variable *dir* of type *Direction_d<Kernel>* initialized to the direction of the i-th base vector of dimension d.
Precondition: $0 \leq i < d$.

Direction_d<Kernel> dir(RT x, RT y);

introduces a variable *dir* of type *Direction_d<Kernel>* in 2-dimensional space.

Direction_d<Kernel> dir(RT x, RT y, RT z);

introduces a variable *dir* of type *Direction_d<Kernel>* in 3-dimensional space.

Operations

int	*dir.dimension()*	returns the dimension of *dir*.
RT	*dir.delta(int i)*	returns the i-th component of *dir*. *Precondition*: $0 \leq i < d$.
RT	*dir*[*int i*]	returns the i-th delta of *dir*. *Precondition*: $0 \leq i < d$.
Delta_const_iterator	*dir.deltas_begin()*	returns an iterator pointing to the first delta of *dir*.
Delta_const_iterator	*dir.deltas_end()*	returns an iterator pointing beyond the last delta of *dir*.
Vector_d<Kernel>	*dir.vector()*	returns a vector pointing in direction *dir*.
bool	*dir.is_degenerate()*	returns true iff *dir.delta(i)==0* for all $0 \leq i < d$.
Direction_d<Kernel>	*dir.transform(Aff_transformation_d<Kernel> t)*	returns $t(p)$.
Direction_d<Kernel>	*dir.opposite()*	returns the direction opposite to *dir*.
Direction_d<Kernel>	$-dir$	returns the direction opposite to *dir*.

Downward compatibility

We provide the operations of the lower dimensional interface *dx()*, *dy()*, *dz()*.

Implementation

Directions are implemented by arrays of integers as an item type. All operations like creation, initialization, tests, inversion, input and output on a direction d take time $O(d.dimension())$. *dimension()*, coordinate access and conversion take constant time. The space requirement is $O(d.dimension())$.

CGAL::Line_d<Kernel>

Definition

An instance of data type *Line_d* is an oriented line in d-dimensional Euclidean space.

Types

Line_d<Kernel>:: LA	the linear algebra layer.

Creation

Line_d<Kernel> l; introduces a variable *l* of type *Line_d<Kernel>*.

Line_d<Kernel> l(Point_d<Kernel> p, Point_d<Kernel> q);

introduces a line through p and q and oriented from p to q.
Precondition: p and q are distinct and have the same dimension.

Line_d<Kernel> l(Point_d<Kernel> p, Direction_d<Kernel> dir);

introduces a line through p with direction *dir*.
Precondition: *p.dimension()==dir.dimension()*, *dir* is not degenerate.

Line_d<Kernel> l(Segment_d<Kernel> s);

introduces a variable *l* of type *Line_d<Kernel>* and initializes it to the line through *s.source()* and *s.target()* with direction from *s.source()* to *s.target()*.
Precondition: *s* is not degenerate.

Line_d<Kernel> l(Ray_d<Kernel> r);

introduces a variable *l* of type *Line_d<Kernel>* and initializes it to the line through *r.point(1)* and *r.point(2)*.

Operations

int	*l.dimension()*	returns the dimension of the ambient space.
Point_d<Kernel>	*l.point(int i)*	returns an arbitrary point on *l*. It holds that *point(i) == point(j)*, iff *i==j*. Furthermore, *l* is directed from *point(i)* to *point(j)*, for all $i < j$.
Line_d<Kernel>	*l.opposite()*	returns the line *(point(2),point(1))* of opposite direction.

Direction_d<Kernel>	*l.direction()*	returns the direction of *l*.
Line_d<Kernel>	*l.transform(Aff_transformation_d<Kernel> t)*	
		returns $t(l)$. *Precondition*: *l.dimension()==t.dimension()*.
Line_d<Kernel>	$l +$ *Vector_d<Kernel> v*	
		returns *l*+*v*, i.e., *l* translated by vector *v*. *Precondition*: *l.dimension()==v.dimension()*.
Point_d<Kernel>	*l.projection(Point_d<Kernel> p)*	
		returns the point of intersection of *l* with the hyperplane that is orthogonal to *l* and that contains *p*. *Precondition*: *l.dimension()==p.dimension()*.
bool	*l.has_on(Point_d<Kernel> p)*	
		returns true if *p* lies on *l* and false otherwise. *Precondition*: *l.dimension()==p.dimension()*.

Non-Member Functions

bool	*weak_equality(l1, l2)*	
		Test for equality as unoriented lines. *Precondition*: *l1.dimension()==l2.dimension()*.
bool	*parallel(l1, l2)*	returns true if *l1* and *l2* are parallel as unoriented lines and false otherwise. *Precondition*: *l1.dimension()==l2.dimension()*.

Implementation

Lines are implemented by a pair of points as an item type. All operations like creation, initialization, tests, direction calculation, input and output on a line *l* take time $O(l.dimension())$. *dimension()*, coordinate and point access, and identity test take constant time. The operations for intersection calculation also take time $O(l.dimension())$. The space requirement is $O(l.dimension())$.

CGAL::Ray_d<Kernel>

Class

Definition

An instance of data type *Ray_d* is a ray in d-dimensional Euclidean space. It starts in a point called the source of r and it goes to infinity.

Types

Ray_d<Kernel>:: LA — the linear algebra layer.

Creation

Ray_d<Kernel> r; — introduces some ray in d-dimensional space.

Ray_d<Kernel> r(Point_d<Kernel> p, Point_d<Kernel> q);

introduces a ray through p and q and starting at p.
Precondition: p and q are distinct and have the same dimension.
Precondition: *p.dimension()==q.dimension()*.

Ray_d<Kernel> r(Point_d<Kernel> p, Direction_d<Kernel> dir);

introduces a ray starting in p with direction *dir*.
Precondition: p and *dir* have the same dimension and *dir* is not degenerate.
Precondition: *p.dimension()==dir.dimension()*.

Ray_d<Kernel> r(Segment_d<Kernel> s);

introduces a ray through *s.source()* and *s.target()* and starting at *s.source()*.
Precondition: s is not degenerate.

Operations

int	*r.dimension()*	returns the dimension of the ambient space.
Point_d<Kernel>	*r.source()*	returns the source point of r.
Point_d<Kernel>	*r.point(int i)*	returns a point on r. *point(0)* is the source. *point(i)*, with $i > 0$, is different from the source. *Precondition*: $i \geq 0$.
Direction_d<Kernel>	*r.direction()*	returns the direction of r.

Line_d<Kernel>	*r.supporting_line()*	returns the supporting line of r.
Ray_d<Kernel>	*r.opposite()*	returns the ray with direction opposite to r and starting in *source*.
Ray_d<Kernel>	*r.transform(Aff_transformation_d<Kernel> t)*	returns $t(r)$. *Precondition*: *r.dimension()==t.dimension()*.
Ray_d<Kernel>	$r +$ *Vector_d<Kernel> v*	returns *r+v*, i.e., r translated by vector v. *Precondition*: *r.dimension()==v.dimension()*.
bool	*r.has_on(Point_d<Kernel> p)*	A point is on r, iff it is equal to the source of r, or if it is in the interior of r. *Precondition*: *r.dimension()==p.dimension()*.

Non-Member Functions

bool	*parallel(r1, r2)*	returns true if the unoriented supporting lines of *r1* and *r2* are parallel and false otherwise. *Precondition*: *r1.dimension()==r2.dimension()*.

Implementation

Rays are implemented by a pair of points as an item type. All operations like creation, initialization, tests, direction calculation, input and output on a ray r take time $O(r.dimension())$. *dimension()*, coordinate and point access, and identity test take constant time. The space requirement is $O(r.dimension())$.

CGAL::Segment_d<Kernel>

Class

Definition

An instance s of the data type *Segment_d* is a directed straight line segment in d-dimensional Euclidean space connecting two points p and q. p is called the source point and q is called the target point of s, both points are called endpoints of s. A segment whose endpoints are equal is called *degenerate*.

Types

Segment_d<Kernel>:: LA — the linear algebra layer.

Creation

Segment_d<Kernel> s;

introduces a variable s of type *Segment_d<Kernel>*.

Segment_d<Kernel> s(Point_d<Kernel> p, Point_d<Kernel> q);

introduces a variable s of type *Segment_d<Kernel>* which is initialized to the segment (p,q).
Precondition: *p.dimension()==q.dimension()*.

Segment_d<Kernel> s(Point_d<Kernel> p, Vector_d<Kernel> v);

introduces a variable s of type *Segment_d<Kernel>* which is initialized to the segment *(p,p+v)*.
Precondition: *p.dimension()==v.dimension()*.

Operations

int	*s.dimension()*	returns the dimension of the ambient space.
Point_d<Kernel>	*s.source()*	returns the source point of segment s.
Point_d<Kernel>	*s.target()*	returns the target point of segment s.
Point_d<Kernel>	*s.vertex(int i)*	returns source or target of s: *vertex(0)* returns the source, *vertex(1)* returns the target. The parameter i is taken modulo 2, which gives easy access to the other vertex. *Precondition*: $i \geq 0$.
Point_d<Kernel>	*s.point(int i)*	returns *vertex(i)*.

Point_d<Kernel>	s[*int i*]	returns *vertex(i)*.
Point_d<Kernel>	*s.min()*	returns the lexicographically smaller vertex.
Point_d<Kernel>	*s.max()*	returns the lexicographically larger vertex.
Segment_d<Kernel>	*s.opposite()*	returns the segment *(target(),source())*.
Direction_d<Kernel>	*s.direction()*	returns the direction from source to target. *Precondition*: *s* is non-degenerate.
Vector_d<Kernel>	*s.vector()*	returns the vector from source to target.
FT	*s.squared_length()*	returns the square of the length of *s*.
bool	*s.has_on(Point_d<Kernel> p)*	
		returns true if *p* lies on *s* and false otherwise. *Precondition*: *s.dimension()==p.dimension()*.
Line_d<Kernel>	*s.supporting_line()*	returns the supporting line of *s*. *Precondition*: *s* is non-degenerate.
Segment_d<Kernel>	*s.transform(Aff_transformation_d<Kernel> t)*	
		returns $t(s)$. *Precondition*: *s.dimension()==t.dimension()*.
Segment_d<Kernel>	$s +$ *Vector_d<Kernel> v*	
		returns $s + v$, i.e., *s* translated by vector *v*. *Precondition*: *s.dimension()==v.dimension()*.
bool	*s.is_degenerate()*	returns true if *s* is degenerate i.e. *s.source()=s.target()*.

Non-Member Functions

bool	*weak_equality(s1, s2)*	
		Test for equality as unoriented segments. *Precondition*: *s1.dimension()==s2.dimension()*.
bool	*parallel(s1, s2)*	return true if one of the segments is degenerate or if the unoriented supporting lines are parallel. *Precondition*: *s1.dimension()==s2.dimension()*.

bool *common_endpoint(s1, s2, Point_d<Kernel>& common)*

if *s1* and *s2* touch in a common end point, this point is assigned to *common* and the result is *true*, otherwise the result is *false*. If *s1==s2* then one of the endpoints is returned.
Precondition: *s1.dimension()==s2.dimension()*.

Implementation

Segments are implemented by a pair of points as an item type. All operations like creation, initialization, tests, the calculation of the direction and source - target vector, input and output on a segment *s* take time $O(s.dimension())$. *dimension()*, coordinate and end point access, and identity test take constant time. The operations for intersection calculation also take time $O(s.dimension())$. The space requirement is $O(s.dimension())$.

CGAL::Hyperplane_d<Kernel>

Definition

An instance of data type *Hyperplane_d* is an oriented hyperplane in d - dimensional space. A hyperplane h is represented by coefficients $(c_0, c_1, \ldots, c_d)$ of type RT. At least one of c_0 to c_{d-1} must be non-zero. The plane equation is $\sum_{0 \leq i < d} c_i x_i + c_d = 0$, where x_0 to x_{d-1} are Cartesian point coordinates. For a particular x the sign of $\sum_{0 \leq i < d} c_i x_i + c_d$ determines the position of a point x with respect to the hyperplane (on the hyperplane, on the negative side, or on the positive side).

There are two equality predicates for hyperplanes. The (weak) equality predicate (*weak_equality*) declares two hyperplanes equal if they consist of the same set of points, the strong equality predicate (*operator==*) requires in addition that the negative halfspaces agree. In other words, two hyperplanes are strongly equal if their coefficient vectors are positive multiples of each other and they are (weakly) equal if their coefficient vectors are multiples of each other.

Types

Hyperplane_d<Kernel>:: LA

the linear algebra layer.

Hyperplane_d<Kernel>:: Coefficient_const_iterator

a read-only iterator for the coefficients.

Creation

Hyperplane_d<Kernel> h;

introduces a variable h of type *Hyperplane_d<Kernel>*.

template <class InputIterator>
Hyperplane_d<Kernel> h(int d, InputIterator first, InputIterator last, RT D);

introduces a variable h of type *Hyperplane_d<Kernel>* initialized to the hyperplane with coefficients *set [first,last)* and D.
Precondition: *size [first,last) == d*.
Requirement: The value type of InputIterator is RT.

template <class InputIterator>
Hyperplane_d<Kernel> h(int d, InputIterator first, InputIterator last);

introduces a variable h of type *Hyperplane_d<Kernel>* initialized to the hyperplane with coefficients *set [first,last)*.
Precondition: *size [first,last) == d+1*.
Requirement: The value type of InputIterator is RT.

template <class ForwardIterator>
Hyperplane_d<Kernel> h(ForwardIterator first, ForwardIterator last, Point_d<Kernel> o, Oriented_side side = ON_ORIENTED_BOUNDARY)

constructs some hyperplane that passes through the points in *set [first,last)*. If *side* is *ON_POSITIVE_SIDE* or *ON_NEGATIVE_SIDE* then *o* is on that side of the constructed hyperplane.
Precondition: A hyperplane with the stated properties must exist.
Requirement: The value type of *ForwardIterator* is *Point_d<Kernel>*.

Hyperplane_d<Kernel> h(Point_d<Kernel> p, Direction_d<Kernel> dir);

constructs the hyperplane with normal direction *dir* that passes through p. The direction *dir* points into the positive side.
Precondition: *p.dimension()==dir.dimension()* and *dir* is not degenerate.

Hyperplane_d<Kernel> h(RT a, RT b, RT c);

introduces a variable *h* of type *Hyperplane_d<Kernel>* in 2-dimensional space with equation $ax+by+c=0$.

Hyperplane_d<Kernel> h(RT a, RT b, RT c, RT d);

introduces a variable *h* of type *Hyperplane_d<Kernel>* in 3-dimensional space with equation $ax+by+cz+d=0$.

Operations

int *h.dimension()* returns the dimension of *h*.

RT *h*[*int i*] returns the *i*-th coefficient of *h*.
Precondition: $0 \leq i \leq d$.

RT *h.coefficient(int i)*

returns the *i*-th coefficient of *h*.
Precondition: $0 \leq i \leq d$.

Coefficient_const_iterator *h.coefficients_begin()*

returns an iterator pointing to the first coefficient.

Coefficient_const_iterator *h.coefficients_end()*

returns an iterator pointing beyond the last coefficient.

Vector_d<Kernel> *h.orthogonal_vector()*

returns the orthogonal vector of h. It points from the negative halfspace into the positive halfspace and its homogeneous coordinates are $(c_0, \ldots, c_{d-1}, 1)$.

Direction_d<Kernel> *h.orthogonal_direction()*

returns the orthogonal direction of h. It points from the negative halfspace into the positive halfspace.

Oriented_side *h.oriented_side(Point_d<Kernel> p)*

returns the side of the hyperplane h containing p.
Precondition: *h.dimension() == p.dimension()*.

bool *h.has_on(Point_d<Kernel> p)*

returns true iff point p lies on the hyperplane h.
Precondition: *h.dimension() == p.dimension()*.

bool *h.has_on_boundary(Point_d<Kernel> p)*

returns true iff point p lies on the boundary of hyperplane h.
Precondition: *h.dimension() == p.dimension()*.

bool *h.has_on_positive_side(Point_d<Kernel> p)*

returns true iff point p lies on the positive side of hyperplane h.
Precondition: *h.dimension() == p.dimension()*.

bool *h.has_on_negative_side(Point_d<Kernel> p)*

returns true iff point p lies on the negative side of hyperplane h.
Precondition: *h.dimension() == p.dimension()*.

Hyperplane_d<Kernel> *h.transform(Aff_transformation_d<Kernel> t)*

returns $t(h)$.
Precondition: *h.dimension() == t.dimension()*.

Non-Member Functions

bool *weak_equality(h1, h2)*

test for weak equality.
Precondition: *h1.dimension() == h2.dimension()*.

Implementation

Hyperplanes are implemented by arrays of integers as an item type. All operations like creation, initialization, tests, vector arithmetic, input and output on a hyperplane h take time $O(h.dimension())$. coordinate access and *dimension()* take constant time. The space requirement is $O(h.dimension())$.

Class

CGAL::Sphere_d<Kernel>

Definition

An instance S of the data type *Sphere_d* is an oriented sphere in some d-dimensional space. A sphere is defined by $d+1$ points (class *Point_d<Kernel>*). We use A to denote the array of the defining points. A set A of defining points is *legal* if either the points are affinely independent or if the points are all equal. Only a legal set of points defines a sphere in the geometric sense and hence many operations on spheres require the set of defining points to be legal. The orientation of S is equal to the orientation of the defining points, i.e., *orientation(A)*.

Types

Sphere_d<Kernel>:: LA — the linear algebra layer.

Sphere_d<Kernel>:: point_iterator

a read-only iterator for points defining the sphere.

Creation

Sphere_d<Kernel> S; — introduces a variable S of type *Sphere_d<Kernel>*.

template <class ForwardIterator>
Sphere_d<Kernel> *S(int d, ForwardIterator first, ForwardIterator last);*

introduces a variable S of type *Sphere_d<Kernel>*. S is initialized to the sphere through the points in A = *tuple [first,last)*.
Precondition: A consists of $d+1$ d-dimensional points.
Requirement: The value type of ForwardIterator is *Point_d<Kernel>*.

Operations

int *S.dimension()* — returns the dimension of the ambient space.

Point_d<Kernel> *S.point(int i)* — returns the ith defining point.
Precondition: $0 \leq i \leq dim$.

point_iterator *S.points_begin()*

returns an iterator pointing to the first defining point.

point_iterator *S.points_end()*

returns an iterator pointing beyond the last defining point.

bool *S.is_degenerate()*

returns true iff the defining points are not full dimensional.

bool *S.is_legal()* returns true iff the set of defining points is legal. A set of defining points is legal iff their orientation is non-zero or if they are all equal.

Point_d<Kernel> *S.center()* returns the center of *S*.
Precondition: *S* is legal.

FT *S.squared_radius()*

returns the squared radius of the sphere.
Precondition: *S* is legal.

Orientation *S.orientation()*

returns the orientation of *S*.

Oriented_side *S.oriented_side(Point_d<Kernel> p)*

returns either the constant *ON_ORIENTED_BOUNDARY*, *ON_POSITIVE_SIDE*, or *ON_NEGATIVE_SIDE*, iff p lies on the boundary, properly on the positive side, or properly on the negative side of sphere, resp.
Precondition: *S.dimension()==p.dimension()*.

Bounded_side *S.bounded_side(Point_d<Kernel> p)*

returns *ON_BOUNDED_SIDE*, *ON_BOUNDARY*, or *ON_UNBOUNDED_SIDE* iff p lies properly inside, on the boundary, or properly outside of sphere, resp.
Precondition: *S.dimension()==p.dimension()*.

bool *S.has_on_positive_side(Point_d<Kernel> p)*

returns *S.oriented_side(p)==ON_POSITIVE_SIDE*.
Precondition: *S.dimension()==p.dimension()*.

bool *S.has_on_negative_side(Point_d<Kernel> p)*

returns *S.oriented_side(p)==ON_NEGATIVE_SIDE*.
Precondition: *S.dimension()==p.dimension()*.

bool *S.has_on_boundary(Point_d<Kernel> p)*

returns *S.oriented_side(p)==ON_ORIENTED_BOUNDARY*, which is the same as *S.bounded_side(p)==ON_BOUNDARY*.
Precondition: *S.dimension()==p.dimension()*.

bool *S.has_on_bounded_side(Point_d<Kernel> p)*

returns *S.bounded_side(p)==ON_BOUNDED_SIDE*.
Precondition: *S.dimension()==p.dimension()*.

bool *S.has_on_unbounded_side(Point_d<Kernel> p)*

returns *S.bounded_side(p)==ON_UNBOUNDED_SIDE*.
Precondition: *S.dimension()==p.dimension()*.

Sphere_d<Kernel>

S.opposite() returns the sphere with the same center and squared radius as *S* but with opposite orientation.

Sphere_d<Kernel>

$S +$ *Vector_d<Kernel> v*

returns the sphere translated by v.
Precondition: *S.dimension()==v.dimension()*.

Non-Member Functions

bool *weak_equality(S1, S2)*

Test for equality as unoriented spheres.
Precondition: *S1.dimension()==S2.dimension()*.

Implementation

Spheres are implemented by a vector of points as a handle type. All operations like creation, initialization, tests, input and output of a sphere s take time $O(s.dimension())$. *dimension()*, point access take constant time. The *center()*-operation takes time $O(d^3)$ on its first call and constant time thereafter. The sidedness and orientation tests take time $O(d^3)$. The space requirement for spheres is $O(s.dimension())$ neglecting the storage room of the points.

CGAL::Iso_box_d<Kernel>

Class

Definition

An object b of the data type *Iso_box_d<Kernel>* is an iso-box in the Euclidean space $\mathbb{E}^d$ with edges parallel to the axes of the coordinate system.

Creation

Iso_box_d<Kernel> b(const Point_d<Kernel>& p, const Point_d<Kernel> &q);

introduces an iso-oriented iso-box b with diagonal opposite vertices p and q.

Operations

bool	*b.operator==(const Iso_box_d<Kernel>& b2) const*	
		Test for equality: two iso-oriented cuboid are equal, iff their lower left and their upper right vertices are equal.
bool	*b.operator!=(const Iso_box_d<Kernel>& b2) const*	
		Test for inequality.
const Point_d<Kernel>&	*b.min() const*	returns the smallest vertex of b.
const Point_d<Kernel>&	*b.max() const*	returns the largest vertex of b.

Predicates

bool	*b.is_degenerate() const*	
		b is degenerate, if all vertices are collinear.
Bounded_side	*b.bounded_side(const Point_d<Kernel>& p) const*	
		returns either *ON_UNBOUNDED_SIDE*, *ON_BOUNDED_SIDE*, or the constant *ON_BOUNDARY*, depending on where point p is.
bool	*b.has_on_boundary(const Point_d<Kernel>& p) const*	
bool	*b.has_on_bounded_side(const Point_d<Kernel>& p) const*	
bool	*b.has_on_unbounded_side(const Point_d<Kernel>& p) const*	

Miscellaneous

Kernel::FT	*b.volume() const*	returns the volume of *b*.

CGAL::Aff_transformation_d<Kernel>

Class

Definition

An instance of the data type *Aff_transformation_d<Kernel>* is an affine transformation of d-dimensional space. It is specified by a square matrix M of dimension $d+1$. All entries in the last row of M except the diagonal entry must be zero; the diagonal entry must be non-zero. A point p with homogeneous coordinates $(p[0],\ldots,p[d])$ can be transformed into the point *p.transform(A)* $= Mp$, where A is an affine transformation created from M by the constructors below.

Types

Aff_transformation_d<Kernel>:: LA

the linear algebra layer.

Aff_transformation_d<Kernel>:: Matrix

the matrix type.

Creation

Aff_transformation_d<Kernel> t;

introduces some transformation.

Aff_transformation_d<Kernel> t(int d, Identity_transformation);

introduces the identity transformation in d-dimensional space.

Aff_transformation_d<Kernel> t(Matrix M);

introduces the transformation of d-space specified by matrix M.
Precondition: M is a square matrix of dimension $d+1$ where entries in the last row of M except the diagonal entry must be zero; the diagonal entry must be non-zero.

template <typename Forward_iterator>
Aff_transformation_d<Kernel> t(Scaling, Forward_iterator start, Forward_iterator end);

introduces the transformation of d-space specified by a diagonal matrix with entries *set [start,end)* on the diagonal (a scaling of the space).
Precondition: *set [start,end)* is a vector of dimension $d+1$.

Aff_transformation_d<Kernel> t(Translation, Vector_d<Kernel> v);

introduces the translation by vector v.

Aff_transformation_d<Kernel> t(int d, Scaling, RT num, RT den);

returns a scaling by a scale factor *num/den*.
Precondition: *den !=0*.

Aff_transformation_d<Kernel> t(int d, Rotation, RT sin_num, RT cos_num, RT den, int e1 = 0, int e2 = 1);

returns a planar rotation with sine and cosine values *sin_num/den* and *cos_num/den* in the plane spanned by the base vectors b_{e1} and b_{e2} in d-space. Thus the default use delivers a planar rotation in the x-y plane.
Precondition: $sin\_num^2 + cos\_num^2 = den^2$ and $0 \leq e_1 < e_2 < d$.
Precondition: *den != 0*

Aff_transformation_d<Kernel> t(int d,
Rotation,
Direction_d<Kernel> dir,
RT num,
RT den,
int e1 = 0,
int e2 = 1)

returns a planar rotation within a two-dimensional linear subspace. The subspace is spanned by the base vectors b_{e1} and b_{e2} in d-space. The rotation parameters are given by the 2-dimensional direction *dir*, such that the difference between the sines and cosines of the rotation given by *dir* and the approximated rotation are at most *num/den* each.

Precondition: *dir.dimension()==2*, *!dir.is_degenerate()* and *num* < *den* is positive, *den != 0*, $0 \leq e_1 < e_2 < d$.

Operations

int	*t.dimension()*	the dimension of the underlying space
Matrix	*t.matrix()*	returns the transformation matrix
Aff_transformation_d<Kernel>	*t.inverse()*	returns the inverse transformation. *Precondition*: *t.matrix()* is invertible.
Aff_transformation_d<Kernel>	$t * s$	composition of transformations. Note that transformations are not necessarily commutative. *t*s* is the transformation which transforms first by *t* and then by *s*.

Implementation

Affine Transformations are implemented by matrices of number type *RT* as a handle type. All operations like creation, initialization, input and output on a transformation *t* take time $O(t.dimension()^2)$. *dimension()* takes constant time. The operations for inversion and composition have the cubic costs of the used matrix operations. The space requirement is $O(t.dimension()^2)$.

12.9 Global Kernel Functions

CGAL::affinely_independent

#include <CGAL/predicates_d.h>

template <class ForwardIterator>
bool *affinely_independent(ForwardIterator first, ForwardIterator last)*

returns true iff the points in A = *tuple [first,last)* are affinely independent.
Precondition: The objects are of the same dimension.
Requirement: The value type of *ForwardIterator* is *Point_d<R>*

Function

Function

CGAL::affine_rank

#include <CGAL/predicates_d.h>

template <class ForwardIterator>
int *affine_rank(ForwardIterator first, ForwardIterator last)*

computes the affine rank of the points in *A* = *tuple [first,last)*.
Precondition: The objects in *A* are of the same dimension.
Requirement: The value type of *ForwardIterator* is *Point_d<R>*.

CGAL::center_of_sphere

#include <CGAL/constructions_d.h>

template <class ForwardIterator>
Point_d<R> *center_of_sphere(ForwardIterator first, ForwardIterator last)*

returns the center of the sphere spanned by the points in A = *tuple[first,last)*.
Precondition: A contains $d+1$ affinely independent points of dimension d.
Requirement: The value type of *ForwardIterator* is *Point_d<R>*.

Function

Function

CGAL::compare_lexicographically

#include <CGAL/predicates_d.h>

Comparison_result *compare_lexicographically(Point_d<R> p, Point_d<R> q)*

Compares the Cartesian coordinates of points p and q lexicographically in ascending order of its Cartesian components *p[i]* and *q[i]* for $i = 0, \ldots, d-1$.
Precondition: *p.dimension() == q.dimension()*

CGAL::contained_in_affine_hull

Function

#include <CGAL/predicates_d.h>

template <class ForwardIterator>

bool	*contained_in_affine_hull(*	*ForwardIterator first,* *ForwardIterator last,* *Point_d<R> p)*

determines whether p is contained in the affine hull of the points in A = *tuple [first,last)*.
Precondition: The objects in A are of the same dimension.
Requirement: The value type of *ForwardIterator* is *Point_d<R>*.

Function

CGAL::contained_in_linear_hull

#include <CGAL/predicates_d.h>

template <class ForwardIterator>
bool contained_in_linear_hull(ForwardIterator first, ForwardIterator last, Vector_d<R> v)

determines whether *v* is contained in the linear hull of the vectors in *A* = *tuple [first,last)*.
Precondition: The objects in *A* are of the same dimension.
Requirement: The value type of *ForwardIterator* is *Vector_d<R>*.

CGAL::contained_in_simplex

#include <CGAL/predicates_d.h>

template <class ForwardIterator>
bool *contained_in_simplex(ForwardIterator first, ForwardIterator last, Point_d<R> p)*

determines whether p is contained in the simplex of the points in A = *tuple [first,last)*.
Precondition: The objects in A are of the same dimension and affinely independent.
Requirement: The value type of *ForwardIterator* is *Point_d<R>*.

Function

CGAL::do_intersect

#include <CGAL/intersections_d.h>

bool *do_intersect(Type1<R> obj1, Type2<R> obj2)*

checks whether *obj1* and *obj2* intersect. Two objects *obj1* and *obj2* intersect if there is a point p that is part of both *obj1* and *obj2*. The intersection region of those two objects is defined as the set of all points p that are part of both *obj1* and *obj2*.
Precondition: the objects are of the same dimension.

The types *Type1* and *Type2* can be any of the following:

- *Point_d<R>*
- *Line_d<R>*
- *Ray_d<R>*
- *Segment_d<R>*
- *Hyperplane_d<R>*

See Also

intersection

CGAL::intersection

Function

#include <CGAL/intersections_d.h>

Object *intersection(Type1<R> f1, Type2<R> f2)*

returns the intersection result of $f1$ and $f2$ by means of the polymorphic wrapper type *Object*. The returned object can be tested for the intersection result and assigned by means of the *object_cast* function.
Precondition: The objects are of the same dimension.

The possible value for types *Type1* and *Type2* and the possible return values wrapped in *Object* are the following:

Type1	Type2	Return Type
Line_d	*Line_d*	*Point_d*, *Line_d*
Segment_d	*Line_d*	*Point_d*, *Segment_d*
Segment_d	*Segment_d*	*Point_d*, *Segment_d*
Ray_d	*Line_d*	*Point_d*, *Ray_d*
Ray_d	*Segment_d*	*Point_d*, *Segment_d*
Ray_d	*Ray_d*	*Point_d*, *Segment_d*, *Ray_d*
Hyperplane_d	*Line_d*	*Point_3*, *Line_3*
Hyperplane_d	*Ray_d*	*Point_d*, *Ray_d*
Hyperplane_d	*Segment_d*	*Point_d*, *Segment_d*

Example

The following example demonstrates the most common use of *intersection* routines.

```
#include <CGAL/intersections_d.h>

template <class R>
void foo(Segment_d<R> seg, Line_d<R> lin)
{
  Object result = intersection(seg, lin);
  if (const Point_d<R> *ipnt = object_cast<Point_d<R> >(&result) ) {
    // handle the point intersection case with *ipnt.
  } else if (const Segment_d<R> *iseg = object_cast<Segment_d<R> >(&result) ) {
    // handle the segment intersection case with *iseg.
  } else {
    // handle the no intersection case.
  }
}
```

See Also

do_intersect, *Kernel::Intersect_d*

CGAL::lexicographically_smaller

#include <CGAL/predicates_d.h>

bool	*lexicographically_smaller(Point_d<R> p, Point_d<R> q)*
	returns *true* iff *p* is lexicographically smaller than *q* with respect to Cartesian lexicographic order of points. *Precondition*: *p.dimension() == q.dimension()*.

Function

Function

CGAL::lexicographically_smaller_or_equal

#include <CGAL/predicates_d.h>

bool	*lexicographically_smaller_or_equal(Point_d<R> p, Point_d<R> q)*	
		returns *true* iff p is lexicographically smaller than q with respect to Cartesian lexicographic order of points or equal to q. *Precondition*: *p.dimension() == q.dimension()*.

CGAL::lift_to_paraboloid

#include <CGAL/constructions_d.h>

Point_d<R> *lift_to_paraboloid(Point_d<R> p)*

returns the projection of $p = (x_0, \ldots, x_{d-1})$ onto the paraboloid of revolution which is the point $(p_0, \ldots, p_{d-1}, \sum_{0 \leq i < d} p_i^2)$ in $(d+1)$-space.

Function

Function

CGAL::linearly_independent

#include <CGAL/predicates_d.h>

template <class ForwardIterator>
bool *linearly_independent(ForwardIterator first, ForwardIterator last)*

decides whether the vectors in A = *tuple [first,last)* are linearly independent.
Precondition: The objects in A are of the same dimension.
Requirement: The value type of *ForwardIterator* is *Vector_d<R>*.

CGAL::linear_base

#include <CGAL/constructions_d.h>

template <class ForwardIterator, class OutputIterator>
OutputIterator *linear_base(ForwardIterator first, ForwardIterator last, OutputIterator result)*

computes a basis of the linear space spanned by the vectors in A = *tuple [first,last)* and returns it via an iterator range starting in *result*. The returned iterator marks the end of the output.
Precondition: A contains vectors of the same dimension d.
Requirement: The value type of *ForwardIterator* and *OutputIterator* is *Vector_d<R>*.

Function

Function

CGAL::linear_rank

#include <CGAL/predicates_d.h>

template <class ForwardIterator>
int *linear_rank(ForwardIterator first, ForwardIterator last)*

computes the linear rank of the vectors in A = *tuple [first,last)*.
Precondition: The objects are of the same dimension.
Requirement: The value type of *ForwardIterator* is *Vector_d<R>*.

CGAL::midpoint

#include <CGAL/constructions_d.h>

Point_d<R>	*midpoint(Point_d<R> p, Point_d<R> q)*
	computes the midpoint of the segment *pq*. *Precondition*: *Precondition*: *p.dimension() == q.dimension()*.

Function

Function

CGAL::orientation

#include <CGAL/predicates_d.h>

template <class ForwardIterator>
Orientation *orientation(ForwardIterator first, ForwardIterator last)*

determines the orientation of the points of the tuple A = *tuple [first,last)* where A consists of $d+1$ points in d-space. This is the sign of the determinant

$$\begin{vmatrix} 1 & 1 & 1 & 1 \\ A[0] & A[1] & \dots & A[d] \end{vmatrix}$$

where $A[i]$ denotes the Cartesian coordinate vector of the i-th point in A.
Precondition: *size [first,last)* == *d+1* and $A[i]$*.dimension()* == d $\forall 0 \le i \le d$.
Requirement: The value type of *ForwardIterator* is *Point_d<R>*.

CGAL::project_along_d_axis

#include <CGAL/constructions_d.h>

Point_d<R>	*project_along_d_axis(Point_d<R> p)*	
		returns p projected along the d-axis onto the hyperspace spanned by the first $d-1$ standard base vectors.

Function

Function

CGAL::side_of_bounded_sphere

#include <CGAL/predicates_d.h>

template <class ForwardIterator>
Bounded_side *side_of_bounded_sphere(ForwardIterator first, ForwardIterator last, Point_d<R> p)*

returns the relative position of point p to the sphere defined by A = *tuple [first,last)*. The order of the points of A does not matter.
Precondition: *orientation(first,last)* is not *ZERO*.
Requirement: The value type of *ForwardIterator* is *Point_d<R>*.

CGAL::side_of_oriented_sphere

#include <CGAL/predicates_d.h>

template <class ForwardIterator>
Oriented_side *side_of_oriented_sphere(ForwardIterator first, ForwardIterator last, Point_d<R> p)*

returns the relative position of point p to the oriented sphere defined by the points in A = *tuple [first,last)* The order of the points in A is important, since it determines the orientation of the implicitly constructed sphere. If the points in A are positively oriented, the positive side is the bounded interior of the sphere.
Precondition: A contains $d+1$ points in d-space.
Requirement: The value type of *ForwardIterator* is *Point_d<R>*.

Function

CGAL::squared_distance

#include <CGAL/constructions_d.h>

FT	*squared_distance(Point_d<R> p, Point_d<R> q)*	
		computes the square of the Euclidean distance between the two points p and q. *Precondition*: The dimensions of p and q are the same.

12.10 Kernel Concept

Kernel_d

The concept of a *kernel* is defined by a set of requirements on the provision of certain types and access member functions to create objects of these types. The types are function object classes to be used within the algorithms and data structures in the basic library of CGAL. This allows you to use any model of a kernel as a traits class in the CGAL algorithms and data structures, unless they require types beyond those provided by a kernel.

Kernel_d subsumes the concept of a *d-dimensional kernel*.

A kernel provides types, construction objects, and generalized predicates. The former replace constructors of the kernel classes and constructive procedures in the kernel. There are also function objects replacing operators, especially for equality testing.

Concept

Types

Kernel_d:: FT	a number type that is a model for *FieldNumberType*
Kernel_d:: RT	a number type that is a model for *RingNumberType*

Coordinate Access

Kernel_d:: Cartesian_const_iterator_d	a type that allows to iterate over the Cartesian coordinates

Geometric Objects

Kernel_d:: Point_d
Kernel_d:: Vector_d
Kernel_d:: Direction_d
Kernel_d:: Hyperplane_d
Kernel_d:: Line_d
Kernel_d:: Ray_d
Kernel_d:: Segment_d
Kernel_d:: Iso_box_d
Kernel_d:: Sphere_d
Kernel_d:: Aff_transformation_d

Constructions

Kernel_d:: Barycentric_coordinates_d	
Kernel_d:: Center_of_sphere_d	a model of *Kernel::Center_of_sphere_d*
Kernel_d:: Compute_coordinate_d	a model of *Kernel::Compute_coordinate_d*
Kernel_d:: Construct_point_d	
Kernel_d:: Construct_vector_d	
Kernel_d:: Construct_direction_d	
Kernel_d:: Construct_hyperplane_d	

Kernel_d:: Construct_segment_d	
Kernel_d:: Construct_iso_box_d	
Kernel_d:: Construct_line_d	
Kernel_d:: Construct_ray_d	
Kernel_d:: Construct_sphere_d	
Kernel_d:: Construct_aff_transformation_d	
Kernel_d:: Construct_cartesian_const_iterator_d	a model of *Kernel::ConstructCartesianConstIterator_d*
Kernel_d:: Intersect_d	a model of *Kernel::Intersect_d*
Kernel_d:: Lift_to_paraboloid_d	a model of *Kernel::Lift_to_paraboloid_d*
Kernel_d:: Linear_base_d	a model of *Kernel::Linear_base_d*
Kernel_d:: Midpoint_d	a model of *Kernel::Midpoint_d*
Kernel_d:: Orthogonal_vector_d	a model of *Kernel::Orthogonal_vector_d*
Kernel_d:: Point_of_sphere_d	a model of *Kernel::Point_of_sphere_d*
Kernel_d:: Point_to_vector_d	a model of *Kernel::Point_to_vector_d*
Kernel_d:: Project_along_d_axis_d	a model of *Kernel::Project_along_d_axis_d*
Kernel_d:: Squared_distance_d	a model of *Kernel::Squared_distance_d*
Kernel_d:: Value_at_d	a model of *Kernel::Value_at_d*
Kernel_d:: Vector_to_point_d	a model of *Kernel::Vector_to_point_d*

Generalized Predicates

Kernel_d:: Affine_rank_d	a model of *Kernel::Affine_rank_d*
Kernel_d:: Affinely_independent_d	a model of *Kernel::Affinely_independent_d*
Kernel_d:: Compare_lexicographically_d	a model of *Kernel::Compare_lexicographically_d*
Kernel_d:: Component_accessor_d	a model of *Kernel::Component_accessor_d*
Kernel_d:: Contained_in_affine_hull_d	a model of *Kernel::Contained_in_affine_hull_d*
Kernel_d:: Contained_in_linear_hull_d	a model of *Kernel::Contained_in_linear_hull_d*
Kernel_d:: Contained_in_simplex_d	a model of *Kernel::Contained_in_simplex_d*
Kernel_d:: Equal_d	a model of *Kernel::Equal_d*
Kernel_d:: Has_on_positive_side_d	a model of *Kernel::Has_on_positive_side_d*
Kernel_d:: Less_coordinate_d	a model of *Kernel::Less_coordinate_d*
Kernel_d:: Less_lexicographically_d	a model of *Kernel::Less_lexicographically_d*
Kernel_d:: Less_or_equal_lexicographically_d	a model of *Kernel::Less_or_equal_lexicographically_d*
Kernel_d:: Linear_rank_d	a model of *Kernel::Linear_rank_d*
Kernel_d:: Linearly_independent_d	a model of *Kernel::Linearly_independent_d*
Kernel_d:: Orientation_d	a model of *Kernel::Orientation_d*
Kernel_d:: Oriented_side_d	a model of *Kernel::Oriented_side_d*
Kernel_d:: Point_dimension_d	a model of *Kernel::Point_dimension_d*
Kernel_d:: Position_on_line_d	
Kernel_d:: Side_of_bounded_sphere_d	a model of *Kernel::Side_of_bounded_sphere_d*
Kernel_d:: Side_of_oriented_sphere_d	a model of *Kernel::Side_of_oriented_sphere_d*

Operations

The following member functions return function objects of the types listed above. The name of the access function is the name of the type returned with an *_object* suffix and no capital letter at the beginning. We only give two examples to show the scheme. For the functors *Construct_point_d* and *Orientation_d* the corresponding functions are:

Kernel::Construct_point_d	*kernel.construct_point_d_object() const*
Kernel::Orientation_d	*kernel.orientation_d_object() const*

Has Models

Cartesian_d<FieldNumberType>, *Homogeneous_d<RingNumberType>*

Kernel::Affinely_independent_d

A model for this must provide:

template <class ForwardIterator>
bool *fo(ForwardIterator first, ForwardIterator last)*

returns true iff the points in A = *tuple [first,last)* are affinely independent.
Precondition: The objects are of the same dimension.
Requirement: The value type of *ForwardIterator* is *Kernel::Point_d*.

Kernel::Affine_rank_d

A model for this must provide:

template <class ForwardIterator>
int *fo(ForwardIterator first, ForwardIterator last)*

computes the affine rank of the points in A = *tuple [first,last)*.
Precondition: The objects are of the same dimension.
Requirement: The value type of *ForwardIterator* is *Kernel::Point_d*.

Kernel::CartesianConstIterator_d

A type representing an iterator to the Cartesian coordinates of a point in d dimensions.

Refines

CopyConstructible, Assignable, DefaultConstructible

Is Model for the Concepts

BidirectionalIterator

See Also

Kernel::ConstructCartesianConstIterator_d . . . . page 1130

Kernel::Center_of_sphere_d

A model for this must provide:

template <class ForwardIterator>
Kernel::Point_d *fo(ForwardIterator first, ForwardIterator last)*

returns the center of the sphere spanned by the points in A = *tuple [first,last)*.
Precondition: A contains $d+1$ affinely independent points of dimension d.
Requirement: The value type of *ForwardIterator* is *Kernel::Point_d*.

Kernel::Compare_lexicographically_d

A model for this must provide:

Comparison_result *fo(Kernel::Point_d p, Kernel::Point_d q)*

Compares the Cartesian coordinates of points p and q lexicographically in ascending order of its Cartesian components *p[i]* and *q[i]* for $i = 0, \ldots, d-1$.
Precondition: The objects are of the same dimension.

Kernel::Component_accessor_d

A model for this must provide:

int *fo.dimension(Kernel::Point_d p)*

returns the dimension of p.

Kernel::RT *fo.homogeneous(Kernel::Point_d p, int i)*

returns the ith homogeneous coordinate of p.
Precondition: *0 <= i <= dimension(p)*.

Kernel::FT *fo.cartesian(Kernel::Point_d p, int i)*

returns the ith Cartesian coordinate of p.
Precondition: *0 <= i < dimension(p)*.

Kernel::Compute_coordinate_d

A model for this must provide:

Kernel::FT *fo(Kernel::Point_d p, int i)*

returns the ith cartesian coordinate of p

Kernel::ConstructCartesianConstIterator_d

A model for this must provide:

Kernel::Cartesian_const_iterator_d

fo(Kernel::Point_d p)

returns an iterator on the 0'th Cartesian coordinate of p.

Kernel::Cartesian_const_iterator_d

fo(Kernel::Point_d p, int)

returns the past the end iterator of the Cartesian coordinates of p.

Refines

AdaptableFunctor (with one argument)

See Also

Kernel::CartesianConstIterator_d . . . . page 1125

Kernel::Contained_in_affine_hull_d

A model for this must provide:

template <class ForwardIterator>
Bounded_side *fo(ForwardIterator first, ForwardIterator last, Kernel::Point_d p)*

determines whether p is contained in the affine hull of the points in A = *tuple [first,last)*.
Precondition: The objects are of the same dimension.
Requirement: The value type of *ForwardIterator* is *Kernel::Point_d*.

Kernel::Contained_in_linear_hull_d

A model for this must provide:

template <class ForwardIterator>
Bounded_side *fo(ForwardIterator first, ForwardIterator last, Kernel::Vector_d v)*

determines whether v is contained in the linear hull of the vectors in A = *tuple [first,last)*.
Precondition: The objects are of the same dimension.
Requirement: The value type of *ForwardIterator* is *Kernel::Vector_d*.

Kernel::Contained_in_simplex_d

A model for this must provide:

template <class ForwardIterator>
Bounded_side *fo(ForwardIterator first, ForwardIterator last, Kernel::Point_d p)*

determines whether p is contained in the simplex of the points in A = *tuple [first,last)*.
Precondition: The objects in A are of the same dimension and affinely independent.
Requirement: The value type of *ForwardIterator* is *Kernel::Point_d*.

Functor
Concept

Kernel::Equal_d

A model for this must provide:

bool *fo(Kernel::Point_d p, Kernel::Point_d q)*

returns true iff p and q are equal (as d-dimensional points).
Precondition: p and q have the same dimension.

Kernel::Has_on_positive_side_d

A model for this must provide:

template <class Kernel_object>
bool *fo(Kernel_object o, Kernel::Point_d p)*

returns true iff p is on the positive side of o. *Kernel_object* may be any of *Kernel::Sphere_d*, *Kernel::Hyperplane_d*.
Precondition: p and o have the same dimension.

Functor
Concept

Kernel::Intersect_d

A model for this must provide:

template <class Kernel_object>
Object *fo(Kernel_object p, Kernel_object q)*

returns the result of the intersection of p and q in form of a polymorphic object. *Kernel_object* may be any of *Kernel::Segment_d*, *Kernel::Ray_d*, *Kernel::Line_d*, *Kernel::Hyperplane_d*.
Precondition: p and q have the same dimension.

Kernel::Less_lexicographically_d

A model for this must provide:

bool *fo(Kernel::Point_d p, Kernel::Point_d q)*

returns *true* iff p is lexicographically smaller than q with respect to Cartesian lexicographic order of points.
Precondition: p and q have the same dimension.

Kernel::Less_or_equal_lexicographically_d

A model for this must provide:

bool	*fo(Kernel::Point_d p, Kernel::Point_d q)*	
		returns *true* iff p is lexicographically smaller than q with respect to Cartesian lexicographic order of points or equal to q. *Precondition*: p and q have the same dimension.

Kernel::Less_coordinate_d

A model for this must provide:

bool	*fo(Kernel::Point_d p, Kernel::Point_d q, int i)*	
		returns *true* iff the ith cartesian coordinate of p is smaller than the ith cartesian coordinate of q. *Precondition*: p and q have the same dimension.

Functor
Concept

Kernel::Lift_to_paraboloid_d

A model for this must provide:

Kernel::Point_d *fo(Kernel::Point_d p)*

returns $p = (x_0, \ldots, x_{d-1})$ lifted to the paraboloid of revolution which is the point $(p_0, \ldots, p_{d-1}, \sum_{0 \le i < d} p_i^2)$ in $(d+1)$-space.

Kernel::Linearly_independent_d

A model for this must provide:

template <class ForwardIterator>
bool *fo(ForwardIterator first, ForwardIterator last)*

decides whether the vectors in A = *tuple [first,last)* are linearly independent.
Precondition: The objects in A are of the same dimension.
Requirement: The value type of *ForwardIterator* is *Kernel_d::Vector_d*.

Functor
Concept

Kernel::Linear_base_d

A model for this must provide:

template <class ForwardIterator, class OutputIterator>
int *fo(ForwardIterator first, ForwardIterator last, OutputIterator result)*

computes a basis of the linear space spanned by the vectors in A = *tuple [first,last)* and returns it via an iterator range starting in *result*. The returned iterator marks the end of the output.
Precondition: A contains vectors of the same dimension d.
Requirement: The value type of *ForwardIterator* and *OutputIterator* is *Kernel::Vector_d*.

Kernel::Linear_rank_d

A model for this must provide:

template <class ForwardIterator>
int *fo(ForwardIterator first, ForwardIterator last)*

computes the linear rank of the vectors in A = *tuple [first,last)*.
Precondition:
Precondition: A contains vectors of the same dimension d.
Requirement: The value type of *ForwardIterator* is *Kernel::Vector_d*.

Functor
Concept

Functor
Concept

Kernel::Midpoint_d

A model for this must provide:

Kernel::Point_d *fo(Kernel::Point_d p, Kernel::Point_d q)*

computes the midpoint of the segment pq.
Precondition: p and q have the same dimension.

Kernel::Orientation_d

A model for this must provide:

template <class ForwardIterator>
Orientation *fo(ForwardIterator first, ForwardIterator last)*

determines the orientation of the points of the tuple A = *tuple [first,last)* where A consists of $d+1$ points in d-space. This is the sign of the determinant

$$\begin{vmatrix} 1 & 1 & 1 & 1 \\ A[0] & A[1] & \ldots & A[d] \end{vmatrix}$$

where *A[i]* denotes the Cartesian coordinate vector of the i-th point in A.
Precondition: *size [first,last)* == *d+1* and *A[i].dimension()* == d $\forall 0 \le i \le d$.
Requirement: The value type of *ForwardIterator* is *Kernel::Point_d*.

Functor
Concept

Kernel::Oriented_side_d

A model for this must provide:

template <class Kernel_object>
Oriented_side *fo(Kernel_object o, Kernel::Point_d p)*

returns the side of *p* with respect to *o*. *Kernel_object* may be any of *Kernel::Sphere_d* or *Kernel::Hyperplane_d*.
Precondition: *p* and *o* have the same dimension.

Kernel::Orthogonal_vector_d

A model for this must provide:

Kernel::Vector_d *fo(Kernel::Hyperplane_d h)*

computes an orthogonal vector to *h*.

Functor
Concept

Kernel::Point_dimension_d

A model for this must provide:

int *fo(Kernel::Point_d p)*

returns the dimension of p

Kernel::Point_of_sphere_d

A model for this must provide:

bool *fo(Kernel::Sphere_d s, int i)*

returns the ith point defining the sphere s.

Functor
Concept

Kernel::Point_to_vector_d

A model for this must provide:

Kernel::Vector_d *fo(Kernel::Point_d p)*

converts p to its geometric vector.

Kernel::Project_along_d_axis_d

A model for this must provide:

Kernel::Point_d *fo(Kernel::Point_d p)*

returns p projected along the d-axis onto the hyperspace spanned by the first $d-1$ standard base vectors.

Kernel::Side_of_bounded_sphere_d

A model for this must provide:

template <class ForwardIterator>
Bounded_side *fo(ForwardIterator first, ForwardIterator last, Kernel::Point_d p)*

returns the relative position of point p to the sphere defined by A = *tuple [first,last)*. The order of the points of A does not matter.
Precondition: *orientation(first,last)* is not *ZERO*.
Requirement: The value type of *ForwardIterator* is *Kernel::Point_d*.

Kernel::Side_of_oriented_sphere_d

A model for this must provide:

template <class ForwardIterator>
Bounded_side *fo(ForwardIterator first, ForwardIterator last, Kernel::Point_d p)*

returns the relative position of point p to the oriented sphere defined by the points in A = *tuple [first,last)* The order of the points in A is important, since it determines the orientation of the implicitly constructed sphere. If the points in A are positively oriented, the positive side is the bounded interior of the sphere.
Precondition: A contains $d+1$ points in d-space.
Requirement: The value type of *ForwardIterator* is *Kernel::Point_d*.

Functor
Concept

Kernel::Squared_distance_d

A model for this must provide:

Kernel::FT *fo(Kernel::Point_d p, Kernel::Point_d q)*

computes the square of the Euclidean distance between the two points p and q.
Precondition: The dimensions of p and q are the same.

Kernel::Value_at_d

A model for this must provide:

Kernel::FT *fo(Kernel::Hyperplane_d h, Kernel::Point_d p)*

computes the value of h evaluated at p.
Precondition: p and h have the same dimension.

Kernel::Vector_to_point_d

A model for this must provide:

Kernel::Point_d *fo(Kernel::Vector_d v)*

converts v to the affine point $0 + v$.

Chapter 13

2D Circular Geometry Kernel

Pedro Machado Manhães de Castro, Sylvain Pion, and Monique Teillaud

Contents

13.1 Introduction

The goal of the circular kernel is to offer to the user a large set of functionalities on circles and circular arcs in the plane. All the choices (interface, robustness, representation, and so on) made here are consistent with the choices made in the CGAL kernel, for which we refer the user to the 2D kernel manual.

In this first release, all functionalities necessary for computing an arrangement of circular arcs and these line segments are defined. Three traits classes are provided for the CGAL arrangement package.

13.2 Software Design

The design is done in such a way that the algebraic concepts and the geometric concepts are clearly separated. *Circular_kernel_2* has therefore two template parameters:

- the first parameter must model the CGAL three dimensional *Kernel* concept. The circular kernel derives from it, and it provides all elementary geometric objects like points, lines, circles, and elementary functionality on them.

- the second parameter is the algebraic kernel, which is responsible for computations on polynomials and algebraic numbers. It has to be a model of concept *AlgebraicKernelForCircles*. The robustness of the package relies on the fact that the algebraic kernel provides exact computations on algebraic objects.

The circular kernel uses the extensibility scheme presented in the 2D kernel manual (see Section 11.5). The types of *Kernel* are inherited by the circular kernel and some types are taken from the *AlgebraicKernelForCircles* parameter. Three new main geometric objects are introduced by *Circular_kernel_2*: circular arcs, points of circular arcs (used in particular for endpoints of arcs and intersection points between arcs) and line segments whose endpoints are points of this new type.

In fact, the circular kernel is documented as a concept, *CircularKernel*, and two models are provided:

- *Circular_kernel_2<Kernel,AlgebraicKernelForCircles>*, the basic kernel,

- and a predefined filtered kernel *Exact_circular_kernel_2*, that is based on similar techniques as *Exact_predicates_exact_constructions_kernel*.

13.3 Examples

The first example shows how to construct circles or circular arcs from points, and how to compute intersections between them using the global function.

```
#include <CGAL/Exact_circular_kernel_2.h>
#include <CGAL/point_generators_2.h>

typedef CGAL::Exact_circular_kernel_2             Circular_k;

typedef CGAL::Point_2<Circular_k>                 Point_2;
typedef CGAL::Circle_2<Circular_k>                Circle_2;
typedef CGAL::Circular_arc_2<Circular_k>          Circular_arc_2;

template <typename T>
double prob_2() {
  CGAL::Random_points_in_square_2<Point_2> g(1.0);
  double prob = 0.0;
  for (int i = 0; i < 10000; i++) {

    Point_2 p1, p2, p3, p4, p5, p6;
    p1 = *g++; p2 = *g++; p3 = *g++;
    p4 = *g++; p5 = *g++; p6 = *g++;

    // the pi's are points inherited from the Cartesian kernel Point_2,
   so,
    // the orientation predicate can be called on them
    if(CGAL::orientation(p1, p2, p3) != CGAL::COUNTERCLOCKWISE)
   std::swap(p1, p3);
    T o1 = T(p1, p2, p3);
    if(CGAL::orientation(p4, p5, p6) != CGAL::COUNTERCLOCKWISE)
   std::swap(p4, p6);
```

```
    T o2 = T(p4, p5, p6);

    std::vector< CGAL::Object > res;
    CGAL::intersection(o1, o2, std::back_inserter(res));

    prob += (res.size() != 0) ? 1.0 : 0.0;
  }
  return prob/10000.0;
}

int main()
{
  std::cout << "What is the probability that two arcs formed by" <<
   std::endl;
  std::cout << "three random counterclockwise-oriented points on" <<
   std::endl;
  std::cout << "an unit square intersect? (wait a second please)" <<
   std::endl;
  std::cout << "The probability is: " << prob_2<Circular_arc_2>() <<
    std::endl << std::endl;

  std::cout << "And what about the probability that two circles formed
   by"
    << std::endl;
  std::cout << "three random counterclockwise-oriented points on" <<
   std::endl;
  std::cout << "an unit square intersect? (wait a second please)" <<
   std::endl;
  std::cout << "The probability is: " << prob_2<Circle_2>() << std::endl;
  return 0;
}
```

File: examples/Circular_kernel_2/intersecting_arcs.cpp

The following example shows how to use a functor of the kernel.

```
#include <CGAL/Exact_circular_kernel_2.h>
#include <CGAL/point_generators_2.h>

typedef CGAL::Exact_circular_kernel_2             Circular_k;

typedef CGAL::Point_2<Circular_k>                 Point_2;
typedef CGAL::Circular_arc_2<Circular_k>          Circular_arc_2;

int main()
{
  int n = 0;
  Circular_arc_2 c = Circular_arc_2(Point_2(10,0), Point_2(5,5),
   Point_2(0, 0));

  for(int i = 0; i <= 10; i++) {
    for(int j = 0; j <= 10; j++) {
      Point_2 p = Point_2(i, j);
```

```
      if(Circular_k().has_on_2_object()(c,p)) {
        n++;
        std::cout << "(" << i << "," << j << ")" << std::endl;
      }
    }
  }
  std::cout << "There are " << n << " points in the [0,..,10]x[0,..,10] "
            << "grid on the circular" << std::endl
            << " arc defined counterclockwisely by the points (0,0),
   (5,5), (10,0)"
            << std::endl << "See the points above." << std::endl;
  return 0;
}
```

File: examples/Circular_kernel_2/functor_has_on_2.cpp

13.4 Design and Implementation History

The first pieces of prototype code were comparisons of algebraic numbers of degree 2, written by Olivier Devillers [DFMT00, DFMT02], and that are still used in the current implementation of *CGAL::Root_of_2*.

Some work was then done in the direction of a "kernel" for CGAL.[1] and the first design emerged in [EKP+04].

The code of this package was initially written by Sylvain Pion and Monique Teillaud who also wrote the manual. Athanasios Kakargias had worked on a prototype version of this kernel in 2003. Julien Hazebrouck participated in the implementation in July and August 2005. The contribution of Pedro Machado Manhães de Castro in summer 2006 improved significantly the efficiency of this kernel. He also added more functionality in 2008.

This work was partially supported by the IST Programme of the EU as a Shared-cost RTD (FET Open) Project under Contract No IST-2000-26473 (ECG - Effective Computational Geometry for Curves and Surfaces) and by the IST Programme of the 6th Framework Programme of the EU as a STREP (FET Open Scheme) Project under Contract No IST-006413 (ACS - Algorithms for Complex Shapes).

[1] Monique Teillaud, First Prototype of a CGAL Geometric Kernel with Circular Arcs, Technical Report ECG-TR-182203-01, 2002
Sylvain Pion and Monique Teillaud, Towards a CGAL-like kernel for curves, Technical Report ECG-TR-302206-01, 2003

2D Circular Geometry Kernel Reference Manual

Pedro Machado Manhães de Castro, Sylvain Pion, and Monique Teillaud

13.5 Geometric Concepts

Functors

13.6 Geometric Kernels and Classes

Kernels

Points

Arcs

13.7 Geometric Global Functions

13.8 Algebraic Concepts

Functors

13.9 Algebraic Kernel and Classes

Kernel

Polynomials

Roots of Polynomials

13.10 Alphabetical List of Reference Pages

Concept

CircularKernel

Refines

Kernel

Has Models

CGAL::Circular_kernel_2<Kernel,AlgebraicKernelForCircles>
CGAL::Exact_circular_kernel_2

Types

A model of *CircularKernel* is supposed to provide some basic types

CircularKernel:: Linear_kernel	Model of *LinearKernel*.
CircularKernel:: Algebraic_kernel	Model of *AlgebraicKernelForCircles*.
CircularKernel:: RT	Model of *RingNumberType*.
CircularKernel:: FT	Model of *FieldNumberType*.
CircularKernel:: Root_of_2	Model of *RootOf_2*.
CircularKernel:: Root_for_circles_2_2	Model of *AlgebraicKernelForCircles::RootForCircles_2_2*.
CircularKernel:: Polynomial_1_2	Model of *AlgebraicKernelForCircles::Polynomial_1_2*.
CircularKernel:: Polynomial_for_circles_2_2	Model of *AlgebraicKernelForCircles::PolynomialForCircles_2_2*.

and to define the following geometric objects

CircularKernel:: Point_2	Model of *Kernel::Point_2*.
CircularKernel:: Circle_2	Model of *Kernel::Circle_2*.
CircularKernel:: Line_arc_2	Model of *CircularKernel::LineArc_2*.
CircularKernel:: Circular_arc_2	Model of *CircularKernel::CircularArc_2*.
CircularKernel:: Circular_arc_point_2	Model of *CircularKernel::CircularArcPoint_2*.

Moreover, a model of *CircularKernel* must provide predicates, constructions and other functionalities.

Predicates

CircularKernel:: Compare_x_2	Model of *CircularKernel::CompareX_2*.
CircularKernel:: Compare_y_2	Model of *CircularKernel::CompareY_2*.
CircularKernel:: Compare_xy_2	Model of *CircularKernel::CompareXY_2*.
CircularKernel:: Equal_2	Model of *CircularKernel::Equal_2*.
CircularKernel:: Compare_y_at_x_2	Model of *CircularKernel::CompareYatX_2*.

CircularKernel:: Compare_y_to_right_2	Model of *CircularKernel::CompareYtoRight_2*.
CircularKernel:: Has_on_2	Model of *CircularKernel::HasOn_2*.
CircularKernel:: Do_overlap_2	Model of *CircularKernel::DoOverlap_2*.
CircularKernel:: Do_intersect_2	Model of *CircularKernel::DoIntersect_2*.
CircularKernel:: Bounded_side_2	Model of *CircularKernel::BoundedSide_2*.
CircularKernel:: Has_on_bounded_side_2	Model of *CircularKernel::HasOnBoundedSide_2*.
CircularKernel:: Has_on_unbounded_side_2	Model of *CircularKernel::HasOnUnboundedSide_2*.
CircularKernel:: In_x_range_2	Model of *CircularKernel::InXRange_2*.
CircularKernel:: Is_vertical_2	Model of *CircularKernel::IsVertical_2*.
CircularKernel:: Is_x_monotone_2	Model of *CircularKernel::IsXMonotone_2*.
CircularKernel:: Is_y_monotone_2	Model of *CircularKernel::IsYMonotone_2*.

Constructions

CircularKernel:: Construct_line_2	Model of *CircularKernel::ConstructLine_2*.
CircularKernel:: Construct_circle_2	Model of *CircularKernel::ConstructCircle_2*.
CircularKernel:: Construct_circular_arc_point_2	Model of *CircularKernel::ConstructCircularArcPoint_2*.
CircularKernel:: Construct_line_arc_2	Model of *CircularKernel::ConstructLineArc_2*.
CircularKernel:: Construct_circular_arc_2	Model of *CircularKernel::ConstructCircularArc_2*.
CircularKernel:: Compute_circular_x_2	Model of *CircularKernel::ComputeCircularX_2*
CircularKernel:: Compute_circular_y_2	Model of *CircularKernel::ComputeCircularY_2*
CircularKernel:: Construct_circular_min_vertex_2	Model of *CircularKernel::ConstructCircularMinVertex_2*.
CircularKernel:: Construct_circular_max_vertex_2	Model of *CircularKernel::ConstructCircularMaxVertex_2*.
CircularKernel:: Construct_circular_source_vertex_2	Model of *CircularKernel::ConstructCircularSourceVertex_2*.

CircularKernel:: Construct_circular_target_vertex_2

Model of *CircularKernel::ConstructCircularTargetVertex_2*.

CircularKernel:: Intersect_2	Model of *CircularKernel::Intersect_2*.
CircularKernel:: Split_2	Model of *CircularKernel::Split_2*.
CircularKernel:: Make_x_monotone_2	Model of *CircularKernel::MakeXMonotone_2*.
CircularKernel:: Make_xy_monotone_2	Model of *CircularKernel::MakeXYMonotone_2*.

Link with the algebraic kernel

CircularKernel:: Get_equation	Model of *CircularKernel::GetEquation*.

Operations

As in the *Kernel* concept, for each of the function objects above, there must exist a member function that requires no arguments and returns an instance of that function object. The name of the member function is the uncapitalized name of the type returned with the suffix *_object* appended. For example, for the function object *CircularKernel::Construct_circular_arc_2* the following member function must exist:

Construct_circular_arc_2

ck.construct_circular_arc_2_object() const

See Also

Kernel . . . . page 581

CGAL::Circular_kernel_2<Kernel,AlgebraicKernelForCircles>

Class

#include <CGAL/Circular_kernel_2.h>

Is Model for the Concepts

CircularKernel

Parameters

The first parameter of the circular kernel must be instantiated with a model of the *Kernel* concept. The *Circular_kernel_2* class template derives from this first parameter, in order to reuse all needed functionalities on basic objects provided by a model of the Kernel concept.

The second parameter, *AlgebraicKernelForCircles*, is meant to provide the circular kernel with all the algebraic functionalities required for the manipulation of algebraic curves.

Inherits From

Kernel

Types

The circular kernel uses basic number types of the algebraic kernel:

typedef AlgebraicKernelForCircles::RT	*RT;*	Ring number type.
typedef AlgebraicKernelForCircles::FT	*FT;*	Field number type.

In fact, the two number types *AlgebraicKernelForCircles::RT* and *Kernel::RT* must coincide, as well as *AlgebraicKernelForCircles::FT* and *Kernel::FT*.

The following types are available, as well as all the functionality on them described in the *CircularKernel* concept.

typedef Line_arc_2<Circular_kernel_2>	*Line_arc_2;*
typedef Circular_arc_2<Circular_kernel_2>	*Circular_arc_2;*
typedef Circular_arc_point_2<Circular_kernel_2>	*Circular_arc_point_2;*

See Also

Kernel .... page 581
AlgebraicKernelForCircles .... page 1223
CGAL::Exact_circular_kernel_2 .... page 1170

Class

CGAL::Exact_circular_kernel_2

#include <CGAL/Exact_circular_kernel_2.h>

Definition

A typedef to a circular kernel that provides both exact geometric predicates and exact geometric constructions. This kernel uses some geometric filtering (based on bounding boxes) to gain efficiency.

Is Model for the Concepts

CircularKernel

See Also

CGAL::Circular_kernel_2<Kernel,AlgebraicKernelForCircles> . . . . page 1169

CircularKernel::CircularArc_2

Concept

Concept for arcs of circles.

Refines

CopyConstructible, Assignable, DefaultConstructible

Has Models

CGAL::Circular_arc_2<CircularKernel>

Concept

CircularKernel::LineArc_2

Definition

Concept for line segments supported by a line that is a model of *Kernel::Line_2* and whose endpoints are models of the *CircularKernel::CircularArcPoint_2* concept.

Refines

CopyConstructible, Assignable, DefaultConstructible

Has Models

CGAL::Line_arc_2<CircularKernel>

CircularKernel::CircularArcPoint_2

Definition

Concept for points on circles, circular arcs or line arcs.

Refines

CopyConstructible, Assignable, DefaultConstructible

Has Models

CGAL::Circular_arc_point_2<CircularKernel>

Class

CGAL::Circular_arc_2<CircularKernel>

#include <CGAL/Circular_arc_2.h>

Is Model for the Concepts

CircularKernel::CircularArc_2

Creation

Circular_arc_2<CircularKernel> ca(Circle_2<CircularKernel> c);

Constructs an arc from a full circle.

Circular_arc_2<CircularKernel> ca(Circle_2<CircularKernel> c,
Circular_arc_point_2<CircularKernel> p,
Circular_arc_point_2<CircularKernel> q)

Constructs the circular arc supported by c, whose source is p and whose target is q when traversing the circle in counterclockwise direction.
Precondition: p and q lie on c.

Circular_arc_2<CircularKernel> ca(Point_2<CircularKernel> p,
Point_2<CircularKernel> q,
Point_2<CircularKernel> r)

Constructs an arc that is supported by the circle of type *Circle_2<CircularKernel>* passing through the points p, q and r. The source and target are respectively p and r, when traversing the supporting circle in the counterclockwise direction. Note that, depending on the orientation of the point triple (p,q,r), q may not lie on the arc.
Precondition: p, q, and r are not collinear.

Access Functions

Circle_2<CircularKernel>	*ca.supporting_circle()*	
Point_2<CircularKernel>	*ca.center() const*	returns the center of the supporting circle.
CircularKernel::FT	*ca.squared_radius() const*	returns the squared radius of the supporting circle.

A circular arc is not oriented. Still, its source and target endpoints can be defined, supposing that its supporting circle in traversed the counterclockwise direction from *source* to *target*.

Circular_arc_point_2<CircularKernel>	*ca.source()*	
Circular_arc_point_2<CircularKernel>	*ca.target()*	

When the methods *source* and *target* return the same point, then the arc is in fact a full circle.

When an arc is x-monotone, its left and right points can be accessed directly:

Circular_arc_point_2<CircularKernel>	*ca.left()*	*Precondition*: *ca.is_x_monotone()*.
Circular_arc_point_2<CircularKernel>	*ca.right()*	*Precondition*: *ca.is_x_monotone()*.
Bbox_2	*ca.bbox() const*	Returns a bounding box containing the arc.

Query Functions

bool	*ca.is_x_monotone()*	Tests whether the arc is x-monotone.
bool	*ca.is_y_monotone()*	Tests whether the arc is y-monotone.

Operations

bool	*ca1 == ca2*	Test for equality. Two arcs are equal, iff their non-oriented supporting circles are equal (i.e. they have same center and same squared radius) and their endpoints are equal.
bool	*ca1 != ca2*	Test for non-equality

I/O

istream&	*std::istream& is >> Circular_arc_2 & ca*
ostream&	*std::ostream& os << Circular_arc_2 ca*

See Also

CGAL::Circular_arc_point_2<CircularKernel> . . . . page 1178
CGAL::Line_arc_2<CircularKernel> . . . . page 1176

CGAL::Line_arc_2<CircularKernel>

#include <CGAL/Line_arc_2.h>

Is Model for the Concepts

CircularKernel::LineArc_2

Creation

Line_arc_2<CircularKernel> la(Line_2<CircularKernel> l, Circular_arc_point_2<CircularKernel> p1, Circular_arc_point_2<CircularKernel> p2)

Construct the line segment supported by l, whose source is $p1$ and whose target is $p2$.
Precondition: $p1$ and $p2$ lie on l.

Line_arc_2<CircularKernel> la(Line_2<CircularKernel> l, Point_2<CircularKernel> p1, Point_2<CircularKernel> p2)

Same.

Line_arc_2<CircularKernel> la(Segment_2<CircularKernel> s);

Access Functions

Line_2<CircularKernel>	*la.supporting_line()*	
Circular_arc_point_2<CircularKernel>	*la.source()*	
Circular_arc_point_2<CircularKernel>	*la.target()*	
Circular_arc_point_2<CircularKernel>	*la.left()*	
Circular_arc_point_2<CircularKernel>	*la.right()*	
Bbox_2	*la.bbox() const*	Returns a bounding box containing the line segment.

Query Functions

bool	*la.is_vertical()*	

Operations

bool	*la1* == *la2*	Test for equality. Two arcs are equal, iff their non-oriented supporting lines are equal (i.e. they contain the same set of points) and their endpoints are equal.
bool	*la1* != *la2*	Test for non-equality.

I/O

istream&	*std::istream& is >> Line_arc_2 & ca*
ostream&	*std::ostream& os << Line_arc_2 ca*

The format for input/output is, for each line arc: a *Line_2* (the supporting line) and two *Circular_arc_point_2* (the two endpoints), under the condition that the endpoints are actually lying on the line.

See Also

CGAL::Circular_arc_point_2<CircularKernel> . . . . page 1178
CGAL::Circular_arc_2<CircularKernel> . . . . page 1174

Class

CGAL::Circular_arc_point_2<CircularKernel>

#include <CGAL/Circular_arc_point_2.h>

Is Model for the Concepts

CircularKernel::CircularArcPoint_2

Creation

Circular_arc_point_2<CircularKernel> p(CircularKernel::Point_2 q);

Circular_arc_point_2<CircularKernel> p(CircularKernel::Root_for_circles_2_2 r);

Access Functions

CircularKernel::Root_of_2	*p.x()*	x-coordinate of the point.
CircularKernel::Root_of_2	*p.y()*	y-coordinate of the point.
Bbox_2	*p.bbox() const*	Returns a bounding box around the point.

Operations

bool	$p == q$	Test for equality. Two points are equal, iff their x and y coordinates are equal.
bool	$p \,!= q$	Test for nonequality.
bool	$p < q$	Returns true iff p is lexicographically smaller than q, i.e. either if $p.x() < q.x()$ or if $p.x() == q.x()$ and $p.y() < q.y()$.
bool	$p > q$	Returns true iff p is lexicographically greater than q.
bool	$p <= q$	Returns true iff p is lexicographically smaller than or equal to q.
bool	$p >= q$	Returns true iff p is lexicographically greater than or equal to q.

I/O

istream&	*std::istream& is >> Circular_arc_point_2 & cp*
ostream&	*std::ostream& os << Circular_arc_point_2 ce*

See Also

CGAL::Circular_arc_2<CircularKernel> . . . . page 1174
CGAL::Line_arc_2<CircularKernel> . . . . page 1176

Functor
Concept

CircularKernel::ConstructLine_2

Refines

Kernel::ConstructLine_2

A model *fo* of this type must provide:

CircularKernel::Line_2   *fo(CircularKernel::LineArc_2)*

Constructs the supporting line of a line segment.

CircularKernel::Line_2   *fo(CircularKernel::Polynomial_1_2)*

Constructs a line from an equation.

See Also

CircularKernel::GetEquation .... page 1211

CircularKernel::ConstructCircle_2

Refines

Kernel::ConstructCircle_2

A model *fo* of this type must provide:

CircularKernel::Circle_2

fo(CircularKernel::CircularArc_2)

Constructs the supporting circle of a circular arc.

CircularKernel::Circle_2

fo(CircularKernel::Polynomial_for_circles_2_2)

Constructs a circle from an equation.

See Also

CircularKernel::GetEquation . . . . page 1211

CircularKernel::ConstructCircularArcPoint_2

A model *fo* of this type must provide:

CircularKernel::Circular_arc_point_2

fo(CircularKernel::Root_for_circles_2_2 r)

CircularKernel::Circular_arc_point_2

fo(CircularKernel::Point_2 p)

CircularKernel::ConstructLineArc_2

A model *fo* of this type must provide:

CircularKernel::Line_arc_2

fo.operator()(CircularKernel::Line_2 l,
CircularKernel::Circular_arc_point_2 p1,
CircularKernel::Circular_arc_point_2 p2)

Constructs the line segment supported by *l*, whose source is *p1* and whose target is *p2*.
Precondition: *p1* and *p2* lie on *l*.

CircularKernel::Line_arc_2

fo(CircularKernel::Segment_2 s)

CircularKernel::Line_arc_2

fo(CircularKernel::Point_2 p1, CircularKernel::Point_2 p2)

CircularKernel::Line_arc_2

fo.operator()(CircularKernel::Line_2 l,
CircularKernel::Circle_2 c1,
bool b1,
CircularKernel::Circle_2 c2,
bool b2)

Constructs the line segment whose supporting line is *l*, whose source endpoint is the b_1^{th} intersection of *l* with *c1*, and whose target endpoint is the b_2^{th} intersection of *l* and *c2*, where intersections are ordered lexicographically.
Precondition: *l* intersects both *c1* and *c2*, and the arc defined by the intersections has non-zero length.

CircularKernel::Line_arc_2

fo.operator()(CircularKernel::Line_2 l,
CircularKernel::Line_2 l1,
CircularKernel::Line_2 l2)

Same, for intersections defined by lines instead of circles.

CircularKernel::ConstructCircularArc_2

A model *fo* of this type must provide:

CircularKernel::Circular_arc_2

fo(CircularKernel::Circle_2 c)

Constructs an arc from a full circle.

CircularKernel::Circular_arc_2

*fo.operator()(CircularKernel::Circle_2 c,
CircularKernel::Circular_arc_point_2 p1,
CircularKernel::Circular_arc_point_2 p2)*

Construct the circular arc supported by *c*, whose source is *p1* and whose target is *p2* when traversing the circle in counterclockwise direction.
Precondition: *p1* and *p2* lie on *c*.

CircularKernel::Circular_arc_2

*fo.operator()(CircularKernel::Circle_2 c,
CircularKernel::Circle_2 c1,
bool b1,
CircularKernel::Circle_2 c2,
bool b2)*

Constructs the unique circular arc whose supporting circle is *c*, and whose source is the intersection of *c* and *c1* with index b_1, and whose target is the intersection of *c* and *c2* of index b_2, where intersections are ordered lexicographically, and when traversing the circle in counterclockwise direction.
Precondition: *c* intersects both *c1* and *c2*, and the arc defined by the intersections has non-zero length.

CircularKernel::Circular_arc_2

*fo.operator()(CircularKernel::Circle_2 c,
CircularKernel::Line_2 l1,
bool b1,
CircularKernel::Line_2 l2,
bool b2)*

Same, for intersections defined by lines instead of circles.

CircularKernel::Circular_arc_2

fo.operator()(CircularKernel::Point_2 p,
CircularKernel::Point_2 q,
CircularKernel::Point_2 r)

Constructs an arc that is supported by the circle of type *CircularKernel::Circle_2* passing through the points p, q and r. The source and target are respectively p and r, when traversing the supporting circle in the counterclockwise direction. Note that, depending on the orientation of the point triple (p,q,r), q may not lie on the arc.
Precondition: p, q, and r are not collinear.

Functor
Concept

CircularKernel::ConstructCircularMinVertex_2

A model *fo* of this type must provide:

CircularKernel::Circular_arc_point_2

fo(CircularKernel::Circular_arc_2 c)

Constructs the x-minimal vertex of c.
Precondition: The arc c is x-monotone.

CircularKernel::Circular_arc_point_2

fo(CircularKernel::Line_arc_2 l)

Same, for a line segment.

CircularKernel::ConstructCircularMaxVertex_2

A model *fo* of this type must provide:

CircularKernel::Circular_arc_point_2

fo(CircularKernel::Circular_arc_2 c)

Constructs the x-maximal vertex of c.
Precondition: The arc c is x-monotone.

CircularKernel::Circular_arc_point_2

fo(CircularKernel::Line_arc_2 l)

Same, for a line segment.

Functor
Concept

CircularKernel::ConstructCircularSourceVertex_2

A model *fo* of this type must provide:

CircularKernel::Circular_arc_point_2

fo(CircularKernel::Circular_arc_2 c)

Constructs the source vertex of *c*.

CircularKernel::Circular_arc_point_2

fo(CircularKernel::Line_arc_2 l)

Same, for a line segment.

CircularKernel::ConstructCircularTargetVertex_2

A model *fo* of this type must provide:

CircularKernel::Circular_arc_point_2

fo(CircularKernel::Circular_arc_2 c)

Constructs the target vertex of *c*.

CircularKernel::Circular_arc_point_2

fo(CircularKernel::Line_arc_2 l)

Same, for a line segment.

Functor
Concept

CircularKernel::ConstructBbox_2

A model *fo* of this type must provide operators to construct a bounding box of geometric objects:

CGAL::Bbox_2 *fo(CircularKernel::Circular_arc_point_2 p)*

CGAL::Bbox_2 *fo(CircularKernel::Line_arc_2 l)*

CGAL::Bbox_2 *fo(CircularKernel::Circular_arc_2 c)*

CircularKernel::CompareX_2

Refines

Kernel::CompareX_2

An object *fo* of this type must provide in addition:

Comparison_result	*fo.operator()(CircularKernel::Circular_arc_point_2 p, CircularKernel::Circular_arc_point_2 q)*	
		Compares the x-coordinates of p and q.

See Also

CircularKernel::CompareY_2 . . . . page 1192
CircularKernel::CompareXY_2 . . . . page 1193
CircularKernel::Equal_2 . . . . page 1200

Functor Concept

Functor Concept

CircularKernel::CompareY_2

Refines

Kernel::CompareY_2

An object *fo* of this type must provide in addition:

Comparison_result	*fo.operator()(CircularKernel::Circular_arc_point_2 p, CircularKernel::Circular_arc_point_2 q)*	

Compares the y-coordinates of p and q.

See Also

CircularKernel::CompareX_2 . . . . page 1191
CircularKernel::CompareXY_2 . . . . page 1193
CircularKernel::Equal_2 . . . . page 1200

CircularKernel::CompareXY_2

Functor Concept

Refines

Kernel::CompareXY_2

An object *fo* of this type must provide in addition:

Comparison_result	*fo.operator()(CircularKernel::Circular_arc_point_2 p, CircularKernel::Circular_arc_point_2 q)*	
		Compares p and q according to the lexicographic ordering on x- and y-coordinates.

See Also

CircularKernel::CompareX_2 . . . . page 1191
CircularKernel::CompareY_2 . . . . page 1192
CircularKernel::Equal_2 . . . . page 1200

Functor
Concept

CircularKernel::CompareYatX_2

An object *fo* of this type must provide two operators that compare a point p and an arc a on the vertical line passing through p.

Comparison_result *fo(CircularKernel::Circular_arc_point_2 p, CircularKernel::Circular_arc_2 a)*

For a circular arc.
Precondition: The arc a must be monotone and p must be in the vertical range of a.

Comparison_result *fo(CircularKernel::Circular_arc_point_2 p, CircularKernel::Line_arc_2 a)*

Same for a segment.

CircularKernel::CompareYtoRight_2

An object *fo* of this type must provide operators that compare vertically two arcs on the right side of a common point p:

Comparison_result *fo.operator()(Circular_kernel_2::Circular_arc_2 a1, Circular_kernel_2::Circular_arc_2 a2, Circular_kernel_2::Circular_arc_point_2 p)*

For two circular arcs.
Precondition: p must be a common point to a_1 and a_2, and a_1 and a_2 must be defined to the right of p.

Comparison_result *fo.operator()(Circular_kernel_2::Line_arc_2 a1, Circular_kernel_2::Line_arc_2 a2, Circular_kernel_2::Circular_arc_point_2 p)*

Same for two segments.

Comparison_result *fo.operator()(Circular_kernel_2::Line_arc_2 a1, Circular_kernel_2::Circular_arc_2 a2, Circular_kernel_2::Circular_arc_point_2 p)*

For a segment and an arc.

Comparison_result *fo.operator()(Circular_kernel_2::Circular_arc_2 a1, Circular_kernel_2::Line_arc_2 a2, Circular_kernel_2::Circular_arc_point_2 p)*

Same as previous.

Functor
Concept

CircularKernel::MakeXMonotone_2

A model *fo* of this type must provide:

template < class OutputIterator >
OutputIterator *fo(CircularKernel::Circular_arc_2 ca, OutputIterator oit)*

Splits the arc *ca* into x-monotone arcs that are returned through the output iterator. Note that, to ensure an easy interface with the *Arrangement_2* package, the arcs are returned as *CGAL::Object*'s (see the *ArrangementTraits_2* concept).

For the sake of completeness, the *operator()* must also be defined for a *Line_arc_2*. In this case, the input line arc itself is the only arc returned through the *OutputIterator*.

See Also

CircularKernel::MakeXYMonotone_2 . . . . page 1197

CircularKernel::MakeXYMonotone_2

A model *fo* of this type must provide:

template < class OutputIterator >
OutputIterator *fo(CircularKernel::Circular_arc_2 ca, OutputIterator oit)*

Splits the arc *ca* into *xy*-monotone arcs that are returned through the output iterator. Note that, to ensure an easy interface with the *Arrangement_2* package, the arcs are returned as *CGAL::Object*'s (see the *ArrangementTraits_2* concept).

For the sake of completeness, the *operator()* must also be defined for a *Line_arc_2*. In this case, the input line arc itself is the only arc returned through the *OutputIterator*.

See Also

CircularKernel::MakeXMonotone_2 . . . . page 1196

CircularKernel::Intersect_2

Refines

Kernel::Intersect_2

A model *fo* of this type must provide:

template < class OutputIterator >
OutputIterator *fo(Type1 obj1, Type2 obj2, OutputIterator intersections)*

Copies in the output iterator the intersection elements between the two objects. *intersections* iterates on elements of type *CGAL::Object*, in lexicographic order.

where *Type_1* and *Type_2* can both be either

- *CircularKernel::Line_2* or
- *CircularKernel::Line_arc_2* or
- *CircularKernel::Circle_2* or
- *CircularKernel::Circular_arc_2*.

Depending on the types *Type_1* and *Type_2*, these elements can be assigned to

- *std::pair<CircularKernel::Circular_arc_point_2, unsigned>*, where the unsigned integer is the multiplicity of the corresponding intersection point between *obj_1* and *obj_2*,
- *CircularKernel::Circular_arc_2* in case of an overlap of two circular arcs,
- *CircularKernel::Line_arc_2* in case of an overlap of two line segments or
- *CircularKernel::Line_2* or *CircularKernel::Circle_2* in case of two equal input lines or circles.

See Also

CGAL::intersection . . . . page 1553

CircularKernel::Split_2

A model *fo* of this type must provide:

void *fo.operator()(CircularKernel::Circular_arc_2 a, CircularKernel::Circular_arc_point_2 p, CircularKernel::Circular_arc_2 &a1, CircularKernel::Circular_arc_2 &a2)*

Splits arc a at point p, which creates arcs $a1$ and $a2$.
Precondition: The point lies on the input arc.

void *fo.operator()(CircularKernel::Line_arc_2 l, CircularKernel::Circular_arc_point_2 p, CircularKernel::Line_arc_2 &l1, CircularKernel::Line_arc_2 &l2)*

Same for a line arc.

Functor
Concept

Functor
Concept

CircularKernel::Equal_2

Definition

Testing equality between objects.

Refines

Kernel::Equal_2

An object *fo* of this type must provide in addition:

bool *fo.operator()(CircularKernel::Circular_arc_point_2 p0, CircularKernel::Circular_arc_point_2 p1)*

For two points.

bool *fo(CircularKernel::Circular_arc_2 a0, CircularKernel::Circular_arc_2 a1)*

For two arcs.

bool *fo(CircularKernel::Line_arc_2 a0, CircularKernel::Line_arc_2 a1)*

For two segments.

For the sake of completeness, the *operator()* must also be defined for a *Line_arc_2* and a *Circular_arc_2* as arguments (in any order), and it always returns *false*.

See Also

CircularKernel::CompareX_2 . . . . page 1191
CircularKernel::CompareY_2 . . . . page 1192
CircularKernel::CompareXY_2 . . . . page 1193

CircularKernel::HasOn_2

Functor
Concept

Definition

To test whether a point lies on a curve.

Refines

Kernel::HasOn_2

An object *fo* of this type must provide:

bool *fo(CircularKernel::Line_2 l, CircularKernel::Circular_arc_point_2 p)*

For a line.

bool *fo(CircularKernel::Circle_2 c, CircularKernel::Circular_arc_point_2 p)*

For a circle.

bool *fo(CircularKernel::Line_arc_2 l, CircularKernel::Circular_arc_point_2 p)*

For a line arc.

bool *fo(CircularKernel::Circular_arc_2 c, CircularKernel::Circular_arc_point_2 p)*

For a circular arc.

CircularKernel::DoOverlap_2

Definition

Testing whether the interiors of two curves overlap.

Refines

Kernel::DoOverlap_2

An object *fo* of this type must provide:

bool	*fo(CircularKernel::Line_arc_2 l0, CircularKernel::Line_arc_2 l1)*
	For two line arcs.
bool	*fo(CircularKernel::Circular_arc_2 a0, CircularKernel::Circular_arc_2 a1)*
	For two circular arcs.

CircularKernel::DoIntersect_2

Functor
Concept

Definition

Testing whether two curves intersect.

Refines

Kernel::DoIntersect_2

An object *fo* of this type must provide:

bool *fo(Type1 obj1, Type2 obj2)*

determines if two geometric objects of type Type1 and Type2 intersect or not.

for all pairs *Type1* and *Type2*, where the types *Type1* and *Type2* can be any of the following:

- *CircularKernel::Line_2*
- *CircularKernel::Line_arc_2*
- *CircularKernel::Circle_2*
- *CircularKernel::Circular_arc_2*

See Also

CGAL::do_intersect . . . . page 730

Functor
Concept

CircularKernel::BoundedSide_2

Refines

Kernel::BoundedSide_2

An object *fo* of this type must provide:

Bounded_side	*fo(CircularKernel::Circle_2 s, CircularKernel::Circular_arc_point_2 p)*

See Also

CircularKernel::HasOnBoundedSide_2 . . . . page 1205
CircularKernel::HasOnUnboundedSide_2 . . . . page 1206

CircularKernel::HasOnBoundedSide_2

Functor
Concept

Refines

Kernel::HasOnBoundedSide_2

An object *fo* of this type must provide:

bool	*fo(CircularKernel::Circle_2 s, CircularKernel::Circular_arc_point_2 p)*

See Also

CircularKernel::BoundedSide_2 . . . . page 1204
CircularKernel::HasOnUnboundedSide_2 . . . . page 1206

Functor
Concept

CircularKernel::HasOnUnboundedSide_2

Refines

Kernel::HasOnUnboundedSide_2

An object *fo* of this type must provide:

bool	*fo(CircularKernel::Circle_2 s, CircularKernel::Circular_arc_point_2 p)*

See Also

CircularKernel::BoundedSide_2 . . . . page 1204
CircularKernel::HasOnBoundedSide_2 . . . . page 1205

CircularKernel::InXRange_2

Definition

To test whether a point lies in the vertical range of a curve.

An object *fo* of this type must provide:

bool	*fo(CircularKernel::Line_arc_2 l, CircularKernel::Circular_arc_point_2 p)*	
		For a line arc.
bool	*fo(CircularKernel::Circular_arc_2 c, CircularKernel::Circular_arc_point_2 p)*	
		For a circular arc. *Precondition*: c is x-monotone.

CircularKernel::IsVertical_2

Refines

Kernel::IsVertical_2

An object *fo* of this type must provide:

bool	*fo(CircularKernel::Line_arc_2 l)*	
		For a line arc.
bool	*fo(CircularKernel::Circular_arc_2 c)*	
		For a circular arc, always returns *false*.

CircularKernel::IsXMonotone_2

An object *fo* of this type must provide:

bool	*fo(CircularKernel::Circular_arc_2 c)*	
		Tests whether the arc is x-monotone.
bool	*fo(CircularKernel::Line_arc_2 l)*	
		For a line arc, always returns *true*.

Functor
Concept

CircularKernel::IsYMonotone_2

An object *fo* of this type must provide:

bool	*fo(CircularKernel::Circular_arc_2 c)*	
		Tests whether the arc is y-monotone.
bool	*fo(CircularKernel::Line_arc_2 l)*	
		For a line arc, always returns *true*.

CircularKernel::GetEquation

A model *fo* of this type must provide:

CircularKernel::Polynomial_1_2

fo(CircularKernel::Line_2 c)

Returns the equation of the line.

CircularKernel::Polynomial_for_circles_2_2

fo(CircularKernel::Circle_2 c)

Returns the equation of the circle.

See Also

CircularKernel::ConstructLine_2 . . . . page 1180
CircularKernel::ConstructCircle_2 . . . . page 1181

CircularKernel::ComputeCircularX_2

A model *fo* of this type must provide:

CircularKernel::Root_of_2

fo(CircularKernel::Circular_arc_point_2 p)

Computes the x-coordinate of the point.

CircularKernel::ComputeCircularY_2

A model *fo* of this type must provide:

CircularKernel::Root_of_2

fo(CircularKernel::Circular_arc_point_2 p)

Computes the *y*-coordinate of the point.

CGAL::x_extremal_point

#include <CGAL/global_functions_circular_kernel_2.h>

Circular_arc_point_2<CircularKernel>

x_extremal_point(Circle_2<CircularKernel> c, bool b)

Returns the leftmost (resp. rightmost) point of the circle if *b* is *true* (resp. *false*).

CGAL::y_extremal_point

#include <CGAL/global_functions_circular_kernel_2.h>

Circular_arc_point_2<CircularKernel>

y_extremal_point(Circle_2<CircularKernel> c, bool b)

Returns the bottommost (ressp. topmost) point of the circle if *b* is *true* (resp. *false*).

Function

Function

CGAL::x_extremal_points

#include <CGAL/global_functions_circular_kernel_2.h>

template < class OutputIterator >
OutputIterator *x_extremal_points(Circle_2<CircularKernel> c, OutputIterator res)*

Copies in the output iterator the x-extremal points of the circle. *res* iterates on elements of type *Circular_arc_point_2<CircularKernel>*, sorted in x.

CGAL::y_extremal_points

#include <CGAL/global_functions_circular_kernel_2.h>

template < class OutputIterator >
OutputIterator *y_extremal_points(Circle_2<CircularKernel> c, OutputIterator res)*

Copies in the output iterator the *y*-extremal points of the circle. *res* iterates on elements of type *Circular_arc_point_2<CircularKernel>*, sorted in *y*.

Function

Function

CGAL::has_on

#include <CGAL/global_functions_circular_kernel_2.h>

bool	*has_on(Circle_2<CircularKernel> c, Circular_arc_point_2<CircularKernel> p)*
	Checks whether the point lies on the circle.

CGAL::compare_y_to_right

CGAL::Comparison_result

compare_y_to_right(Circular_arc_2<CircularKernel> ca1,
Circular_arc_2<CircularKernel> ca2,
Circular_arc_point_2<CircularKernel> &p)

Compares vertically the two arcs, to the right of the point p,
Precondition: p is an intersection point of the arcs, and the arcs are defined to the right of p.

Function

Function

CGAL::has_in_x_range

#include <CGAL/global_functions_circular_kernel_2.h>

bool *has_in_x_range(Circular_arc_2<CircularKernel> ca, Circular_arc_point_2<CircularKernel> p)*

Checks whether the point lies in the vertical range defined by the arc.

bool *has_in_x_range(Line_arc_2<CircularKernel> ca, Circular_arc_point_2<CircularKernel> p)*

Checks whether the point lies in the vertical range defined by the line segment.

CGAL::make_x_monotone

#include <CGAL/global_functions_circular_kernel_2.h>

template < class OutputIterator >
OutputIterator *make_x_monotone(Circular_arc_2<CircularKernel> ca, OutputIterator res)*

Copies in the output iterator the x-monotone sub-arcs of *ca*.

Function

Function

CGAL::make_xy_monotone

#include <CGAL/global_functions_circular_kernel_2.h>

template < class OutputIterator >
OutputIterator *make_xy_monotone(Circular_arc_2<CircularKernel> ca, OutputIterator res)*

Copies in the output iterator the *xy*-monotone sub-arcs of *ca*.

AlgebraicKernelForCircles

Concept

Definition

The *AlgebraicKernelForCircles* concept is meant to provide the curved kernel with all the algebraic functionalities required for the manipulation of circular arcs.

Has Models

Algebraic_kernel_for_circles_2_2

Types

A model of *AlgebraicKernelForCircles* is supposed to provide

AlgebraicKernelForCircles:: RT	A model of *RingNumberType*.
AlgebraicKernelForCircles:: FT	A model of *FieldNumberType<RT>*.
AlgebraicKernelForCircles:: Polynomial_1_2	A model of *AlgebraicKernelForCircles::Polynomial_1_2*, for bivariate polynomials of degree up to 1.
AlgebraicKernelForCircles:: Polynomial_for_circles_2_2	A model of *AlgebraicKernelForCircles::PolynomialForCircles_2_2*, for bivariate polynomials of degree up to 2 that can store equations of circles.
AlgebraicKernelForCircles:: Root_of_2	A model of *RootOf_2*, for algebraic numbers of degree up to 2.
AlgebraicKernelForCircles:: Root_for_circles_2_2	A model of *AlgebraicKernelForCircles::RootForCircles_2_2*, for solutions of systems of two models of *AlgebraicKernelForCircles::PolynomialForCircles_2_2*.
AlgebraicKernelForCircles:: Construct_polynomial_1_2	A model of *AlgebraicKernelForCircles::ConstructPolynomial_1_2*.
AlgebraicKernelForCircles:: Construct_polynomial_for_circles_2_2	A model of *AlgebraicKernelForCircles::ConstructPolynomialForCircles_2_2*.
AlgebraicKernelForCircles:: Compare_x	A model of the concept *AlgebraicKernelForCircles::CompareX*.

AlgebraicKernelForCircles:: Compare_y	A model of the concept *AlgebraicKernelForCircles::CompareY*.
AlgebraicKernelForCircles:: Compare_xy	A model of the concept *AlgebraicKernelForCircles::CompareXY*.
AlgebraicKernelForCircles:: Sign_at	A model of the concept *AlgebraicKernelForCircles::SignAt*.
AlgebraicKernelForCircles:: X_critical_points	A model of the concept *AlgebraicKernelForCircles::XCriticalPoints*.
AlgebraicKernelForCircles:: Y_critical_points	A model of the concept *AlgebraicKernelForCircles::YCriticalPoints*.
AlgebraicKernelForCircles:: Solve	A model of the concept *AlgebraicKernelForCircles::Solve*.

See Also

CircularKernel . . . . page 1166
CGAL::Circular_kernel_2<Kernel,AlgebraicKernelForCircles> . . . . page 1169

CGAL::Algebraic_kernel_for_circles_2_2<RT>

#include <CGAL/Algebraic_kernel_for_circles_2_2.h>

Is Model for the Concepts

AlgebraicKernelForCircles

Class

AlgebraicKernelForCircles::RootForCircles_2_2

Definition

Concept to represent the roots of a system of two equations of degree 2 in two variables x and y that are models of concept *AlgebraicKernelForCircles::PolynomialForCircles_2_2*

Operations

The comparison operator == must be provided.

bool	$p == q$

Has Models

CGAL::Root_for_circles_2_2

See Also

AlgebraicKernelForCircles . . . . page 1223

CGAL::Root_for_circles_2_2<FT>

#include <CGAL/Root_for_circles_2_2.h>

Is Model for the Concepts

AlgebraicKernelForCircles::RootForCircles_2_2

Class

AlgebraicKernelForCircles::Polynomial_1_2

Definition

Concept to represent bivariate polynomials of degree 1 whose coefficients are of a type that is a model of the concept *RingNumberType*.

Refines

CopyConstructible, Assignable, DefaultConstructible

Has Models

CGAL::Polynomial_1_2

See Also

AlgebraicKernelForCircles . . . . page 1223

CGAL::Polynomial_1_2<RT>

#include <CGAL/Polynomials_1_2.h>

Is Model for the Concepts

AlgebraicKernelForCircles::Polynomial_1_2

Concept

AlgebraicKernelForCircles::PolynomialForCircles_2_2

Definition

Concept to represent bivariate polynomials of degree up to 2 capable of storing equations of circles, whose center's coordinates, as well as the square of the radius, are of a type that is a model of the concept *FieldNumberType*.

Refines

CopyConstructible, Assignable, DefaultConstructible

Creation

Operations

The comparison operator == must be provided.

bool *AlgebraicKernelForCircles:: PolynomialForCircles_2_2 p == q*

Has Models

CGAL::Polynomial_for_circles_2_2

See Also

AlgebraicKernelForCircles . . . . page 1223

CGAL::Polynomial_for_circles_2_2<FT>

#include <CGAL/Polynomials_2_2.h>

Is Model for the Concepts

AlgebraicKernelForCircles::PolynomialForCircles_2_2

See Also

CGAL::Sqrt_extension<NT,ROOT> . . . . page 220
AlgebraicKernelForCircles . . . . page 1223

Class

AlgebraicKernelForCircles::CompareX

A model *fo* of this type must provide:

template < class OutputIterator >
CGAL::Comparison_result

fo.operator()(AlgebraicKernelForCircles::Root_for_circles_2_2 r1, AlgebraicKernelForCircles::Root_for_circles_2_2 r2)

Compares the x (first) variables of two *Root_for_circles_2_2*.

See Also

AlgebraicKernelForCircles::CompareY . . . . page 1233
AlgebraicKernelForCircles::CompareXY . . . . page 1234
CircularKernel::CompareX_2 . . . . page 1191

AlgebraicKernelForCircles::CompareY

Definition

A model *fo* of this type must provide:

template < class OutputIterator >
CGAL::Comparison_result

fo.operator()(AlgebraicKernelForCircles::Root_for_circles_2_2 r1,
AlgebraicKernelForCircles::Root_for_circles_2_2 r2)

Compares the y (second) variables of two *Root_for_circles_2_2*.

See Also

AlgebraicKernelForCircles::CompareX . . . . page 1232
AlgebraicKernelForCircles::CompareXY . . . . page 1234
CircularKernel::CompareY_2 . . . . page 1192

Concept

AlgebraicKernelForCircles::CompareXY

Definition

A model *fo* of this type must provide:

template < class OutputIterator >
CGAL::Comparison_result

fo.operator()(AlgebraicKernelForCircles::Root_for_circles_2_2 r1, AlgebraicKernelForCircles::Root_for_circles_2_2 r2)

Compares two *Root_for_circles_2_2* lexicographically.

See Also

AlgebraicKernelForCircles::CompareX . . . . page 1232
AlgebraicKernelForCircles::CompareY . . . . page 1233
CircularKernel::CompareXY_2 . . . . page 1193

AlgebraicKernelForCircles::SignAt

Definition

A model *fo* of this type must provide:

template < class OutputIterator >
CGAL::Sign *fo.operator()(AlgebraicKernelForCircles::Polynomial_1_2 p, AlgebraicKernelForCircles::Root_for_circles_2_2 r)*

Computes the sign of polynomial *p* evaluated at a root *r*.

template < class OutputIterator >
CGAL::Sign *fo.operator()(AlgebraicKernelForCircles::Polynomial_for_circles_2_2 p, AlgebraicKernelForCircles::Root_for_circles_2_2 r)*

Same as previous.

Concept

Concept

AlgebraicKernelForCircles::ConstructPolynomial_1_2

A model *fo* of this type must provide:

AlgebraicKernelForCircles::Polynomial_1_2

fo.operator()(AlgebraicKernelForCircles::RT a,
AlgebraicKernelForCircles::RT b,
AlgebraicKernelForCircles::RT c)

Constructs polynomial $ax+by+c$.

See Also

CircularKernel::ConstructLine_2 . . . . page 1180
CircularKernel::GetEquation . . . . page 1211

AlgebraicKernelForCircles::ConstructPolynomialForCircles_2_2

A model *fo* of this type must provide:

AlgebraicKernelForCircles::PolynomialForCircles_2_2

fo.operator()(const AlgebraicKernelForCircles::FT a,
const AlgebraicKernelForCircles::FT b,
const AlgebraicKernelForCircles::FT rsq)

Constructs polynomial *(x-a)*ˆ*2* + *(y-b)*ˆ*2* - *rsq*.

See Also

CircularKernel::ConstructCircle_2 . . . . page 1181
CircularKernel::GetEquation . . . . page 1211

Concept

AlgebraicKernelForCircles::Solve

Definition

A model *fo* of this type must provide:

template < class OutputIterator >
OutputIterator *fo.operator()(AlgebraicKernelForCircles::Polynomial_1_2 p1, AlgebraicKernelForCircles::Polynomial_1_2 p2, OutputIterator res)*

Copies in the output iterator the common roots of *p1* and *p2*, with their multiplicity, as objects of type *std::pair< AlgebraicKernelForCircles::Root_for_circles_2_2, int>*.

template < class OutputIterator >
OutputIterator *fo.operator()(AlgebraicKernelForCircles::Polynomial_1_2 p1, AlgebraicKernelForCircles::Polynomial_for_circles_2_2 p2, OutputIterator res)*

Same as previous.

template < class OutputIterator >
OutputIterator *fo.operator()(AlgebraicKernelForCircles::Polynomial_for_circles_2_2 p1, AlgebraicKernelForCircles::Polynomial_1_2 p2, OutputIterator res)*

Same as previous.

template < class OutputIterator >
OutputIterator *fo.operator()(AlgebraicKernelForCircles::Polynomial_for_circles_2_2 p1, AlgebraicKernelForCircles::Polynomial_for_circles_2_2 p2, OutputIterator res)*

Same as previous.

AlgebraicKernelForCircles::XCriticalPoints

Definition

A model *fo* of this type must provide:

template < class OutputIterator >
OutputIterator *fo.operator()(AlgebraicKernelForCircles::Polynomial_for_circles_2_2 p, OutputIterator res)*

Copies in the output iterator the *x*-critical points of polynomial *p*, as objects of type *AlgebraicKernelForCircles::Root_for_circles_2_2*.

template < class OutputIterator >
AlgebraicKernelForCircles::Root_for_circles_2_2

fo(AlgebraicKernelForCircles::Polynomial_for_circles_2_2 p, bool b)

Computes the *x*-critical point with smallest (resp. largest) *x* of polynomial *p* if *b* is *true* (resp. *false*).

See Also

AlgebraicKernelForCircles::YCriticalPoints . . . . page 1240

Concept

AlgebraicKernelForCircles::YCriticalPoints

Definition

A model *fo* of this type must provide:

template < class OutputIterator >
OutputIterator *fo.operator()(AlgebraicKernelForCircles::Polynomial_for_circles_2_2 p, OutputIterator res)*

Copies in the output iterator the y-critical points of polynomial p, as objects of type *AlgebraicKernelForCircles::Root_for_circles_2_2*.

template < class OutputIterator >
AlgebraicKernelForCircles::Root_for_circles_2_2

fo(AlgebraicKernelForCircles::Polynomial_for_circles_2_2 p, bool i)

Computes the y-critical point with smallest (resp. largest) y of polynomial p if b is *true* (resp. *false*).

See Also

AlgebraicKernelForCircles::XCriticalPoints . . . . page 1239

Chapter 14

3D Spherical Geometry Kernel

Pedro Machado Manhães de Castro, Frédéric Cazals, Sébastien Loriot, and Monique Teillaud

Contents

14.1 Introduction

The goal of the 3D spherical kernel is to offer to the user a large set of functionalities on spheres, circles and circular arcs, in the 3D space or restricted on a given sphere. These functionalities require computing on algebraic numbers, which motivates the creation of a new kernel concept extending the CGAL *Kernel* concept, that is restricted to objects and functionality in a *FieldNumberType*.

All the choices (interface, robustness, representation, and so on) made here are consistent with the choices made in the CGAL kernel, for which we refer the user to the kernel manual (Chapter 11).

14.2 Spherical Kernel Objects

New main geometric objects are introduced by *Spherical_kernel_3*: circular arcs ((model of *SphericalKernel::CircularArc_3*), points of circular arcs (model of *SphericalKernel::CircularArcPoint_3*), and line segments (model of *SphericalKernel::LineArc_3*) whose endpoints are points of this new type.

SphericalKernel::CircularArcPoint_3 is used in particular for endpoints of arcs and intersection points between spheres, circles or arcs. The coordinates of these points are algebraic numbers of degree two. Therefore, general predicates offered by the *Kernel* on *Point_3*, which have coordinates in a *FieldNumberType*, would require heavy algebraic computations in algebraic extensions of higher degrees and thus are not provided on them, which explains the need for a new point type.

A consistent set of predicates and constructions is offered on these new types.

General functionalities

The spherical kernel currently implements a set of fundamental functionalities like intersection, comparisons, inclusion, etc. More might be provided in the future, as long as only algebraic numbers of degree two are used.

Functionalities relative to a sphere

The interface of the underlying objects is extended by providing additional operations that make sense only if the objects are considered on the same sphere. For example, the result of the comparison of the cylindrical or spherical coordinates of two points is well-defined only when looking at them on a given common sphere. The presentation of these operations requires the following definitions:

Coordinate system. Let consider a sphere with center c and radius r. Using the Cartesian frame centered at c, we define a cylindrical coordinate system (θ, z) on that sphere, with $\theta \in [0, 2\pi)$ and $z \in [-r, r]$. θ is given in radian and measured in the xy-plane around the z-axis, starting from $x > 0$, $y = 0$. The z-extremal points of a sphere are its North and South poles defined as (θ, r) and $(\theta, -r)$ respectively, for any value of θ. Observe that each point on the sphere different from a pole corresponds to a unique pair (θ, z).

Definition of a meridian. Given a sphere and its associated cylindrical coordinate system, a meridian of that sphere is a circular arc consisting of the points having the same theta-coordinate (the poles are the end points). A plane containing the two poles of that sphere defines two meridians, one on each side of the line passing through the poles. A vector M whose direction is different from that of the latter line defines a unique meridian on that sphere. The plane of that meridian is defined by the direction of M and the two poles. The sense of M disambiguates the choice among the pair of meridians thus defined. On Fig. 14.1, the normal vectors n_0 and n_1 define two meridians of S: the circular arcs A_0 and A_1 respectively.

Types of circles on a sphere. Given a sphere, a circle on that sphere is termed *polar* if it goes through only one pole, *bipolar* if it goes through the two poles of that sphere and *threaded* if it separates the sphere into two connected components, each containing one pole. Any other circle is termed *normal*. These definitions are illustrated on Fig. 14.2.

θ-*extremal points.* Given a sphere one has: a θ-*extremal point* of a normal circle is a point of tangency between the circle and a meridian anchored at the poles of that sphere. Each normal circle defines two such points; the θ-extremal point of a polar circle is the pole the circle goes through. No such point is defined on a bipolar or a threaded circle. These definitions are illustrated on Fig. 14.2. Notice that the θ-extremal points should not be confused with the endpoints of an arbitrary arc on a sphere.

The θ-coordinate of a θ-extremal point of a normal circle on a sphere is well defined. For a polar circle on a sphere, the plane containing the two poles and which is tangent to that circle contains two different meridians. The θ-values of these meridians are the two θ-coordinates associated to the same θ-extremal point of a polar circle.

θ-*monotone circular arcs.* An arc on a sphere is said to be θ-*monotone* if any meridian on that sphere intersects that arc in at most one point. With this definition, a circular arc on a threaded circle is always θ-monotone, and an arc on a polar or normal circle is θ-monotone if it does not contain a θ-extremal point, unless it is an

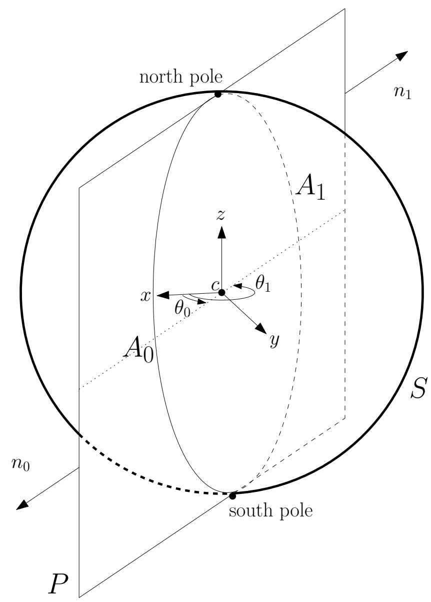


Figure 14.1: Definition of two meridians on S, a sphere of center c. The intersection of the plane P (passing through the two poles of S) and the sphere S is a circle. The two poles of S split that circle into two circular arcs A_0 and A_1, each being a meridian of S. The θ-coordinates of meridians A_0 and A_1 are θ_0 and $\theta_1 = \theta_0 + \pi$ respectively.

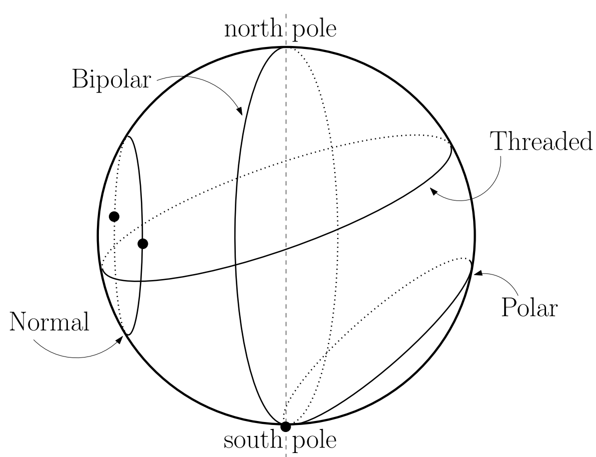


Figure 14.2: The four types of circles on a sphere. Black dots are the θ-extremal points.

endpoint. No such arc is defined on a bipolar circle.

14.3 Software Design

The design of *Spherical_kernel_3* is similar to the design of *Circular_kernel_2* (see Chapter 13). It has two template parameters:

- the first parameter must model the CGAL three dimensional *Kernel* concept. The spherical kernel derives from it, and it provides all elementary geometric objects like points, lines, spheres, circles and elementary functionality on them.
- the second parameter is the algebraic kernel, which is responsible for computations on polynomials and algebraic numbers. It must model the concept *AlgebraicKernelForSpheres*. The robustness of the package relies on the fact that the algebraic kernel provides exact computations on algebraic objects.

The 3D spherical kernel uses the extensibility scheme presented in the kernel manual (see Section 11.5). The types of *Kernel* are inherited by the 3D spherical kernel and some types are taken from the *AlgebraicKernelForSpheres* parameter. *Spherical_kernel_3* introduces new geometric objects as mentioned in Section 14.2.

In fact, the spherical kernel is documented as a concept, *SphericalKernel* and two models are provided:

- *Spherical_kernel_3<Kernel,AlgebraicKernelForSpheres>*, the basic kernel,
- and a predefined kernel *Exact_spherical_kernel_3*.

14.4 Examples

The first example shows how to construct spheres and compute intersections on them using the global function.

```
#include <CGAL/Exact_spherical_kernel_3.h>
#include <CGAL/Random.h>

typedef CGAL::Exact_spherical_kernel_3         Spherical_k;

typedef CGAL::Point_3<Spherical_k>             Point_3;
typedef CGAL::Sphere_3<Spherical_k>            Sphere_3;

int main() {

  CGAL::Random generatorOfgenerator;
  int random_seed = generatorOfgenerator.get_int(0, 123456);
  CGAL::Random theRandom(random_seed);
  int count = 0;

  std::cout << "We will compute the approximate probability that 3
    spheres wit"
  << "h radius 1 intersect on a 5x5x5 box, it might take some time." <<
    std::endl;

  for(int i=0; i<10000; i++) {

    double x1 = theRandom.get_double(0.0,5.0);
    double y1 = theRandom.get_double(0.0,5.0);
```

```
    double z1 = theRandom.get_double(0.0,5.0);
    double r = 1.0;
    double x2 = theRandom.get_double(0.0,5.0);
    double y2 = theRandom.get_double(0.0,5.0);
    double z2 = theRandom.get_double(0.0,5.0);
    double x3 = theRandom.get_double(0.0,5.0);
    double y3 = theRandom.get_double(0.0,5.0);
    double z3 = theRandom.get_double(0.0,5.0);

    Sphere_3 s1 = Sphere_3(Point_3(x1,y1,z1), r);
    Sphere_3 s2 = Sphere_3(Point_3(x2,y2,z2), r);
    Sphere_3 s3 = Sphere_3(Point_3(x3,y3,z3), r);

    std::vector< CGAL::Object > intersecs;
    CGAL::intersection(s1, s2, s3, std::back_inserter(intersecs));
    if(intersecs.size() > 0) count++;
  }

  std::cout << "The approximate probability that 3 spheres with radius 1"
            << std::endl;
  std::cout << "choosen (uniformly) randomly on a 5x5x5 box intersect is:
    "
            << ((double)count)/((double)(10000)) << std::endl;

  return 0;
}
```

File: examples/Circular_kernel_3/intersecting_spheres.cpp

The second example illustrates the use of a functor.

```
#include <CGAL/Exact_spherical_kernel_3.h>
#include <CGAL/Random.h>

typedef CGAL::Exact_spherical_kernel_3         Spherical_k;

typedef CGAL::Point_3<Spherical_k>             Point_3;
typedef CGAL::Circular_arc_3<Spherical_k>      Circular_arc_3;

int main()
{
  int n = 0;
  Circular_arc_3 c = Circular_arc_3(Point_3(10,10,0), Point_3(5,5,5),
   Point_3(0, 0, 0));
  for(int i = 0; i <= 10; i++) {
    for(int j = 0; j <= 10; j++) {
      for(int k = 0; k <= 10; k++) {
        Point_3 p = Point_3(i, j, k);
        if(Spherical_k().has_on_3_object()(c,p)) {
          n++;
          std::cout << "(" << i << "," << j << "," << k << ")" <<
   std::endl;
        }
```

```
      }
    }
  }

  std::cout << "There are " << n << " points in the "
            << "[0,..,10]x[0,..,10]x[0,...,10] "
            << "grid on the circular" << std::endl
            << " arc defined by the points (10,10,0), (5,5,5), (0,0,0)"
            << std::endl << "See the points above." << std::endl;
  return 0;
}
```

File: examples/Circular_kernel_3/functor_has_on_3.cpp

The third example illustrates the use of a functor on objects on the same sphere. The intersection points of two circles on the same sphere are computed and their cylindrical coordinates are then compared.

```
#include <CGAL/Exact_spherical_kernel_3.h>

typedef CGAL::Exact_spherical_kernel_3              SK;

int main(){
  //construction of 3 spheres from their centers and squared radii
  SK::Sphere_3 s1(SK::Point_3(0,0,0),2);
  SK::Sphere_3 s2(SK::Point_3(0,1,0),1);
  SK::Sphere_3 s3(SK::Point_3(1,0,0),3);

  //construct two circles lying on sphere s1
  SK::Circle_3 C1(s1,s2);
  SK::Circle_3 C2(s1,s3);

  SK::Intersect_3 inter;
  //create a functor to compare theta-coordinates on sphere s1
  SK::Compare_theta_z_3 cmp(s1);
  std::vector< CGAL::Object > intersections;
  inter(C1,C2,std::back_inserter(intersections));

  //unsigned integer indicates multiplicity of intersection point
  std::pair<SK::Circular_arc_point_3,unsigned> p1=
    CGAL::object_cast< std::pair<SK::Circular_arc_point_3,unsigned>
   >(intersections[0]);
  std::pair<SK::Circular_arc_point_3,unsigned> p2=
    CGAL::object_cast< std::pair<SK::Circular_arc_point_3,unsigned>
   >(intersections[1]);


  SK::Circular_arc_point_3 t_extreme[2];
  //Compute theta extremal points of circle C1 on sphere s1
  CGAL::theta_extremal_points(C1,s1,t_extreme);

  //The theta coordinates of theta extremal points of C1 enclose that of
   each intersection point.
  assert(cmp(t_extreme[0],p1.first)==CGAL::SMALLER);
```

```
  assert(cmp(t_extreme[0],p2.first)==CGAL::SMALLER);
  assert(cmp(t_extreme[1],p1.first)==CGAL::LARGER);
  assert(cmp(t_extreme[1],p2.first)==CGAL::LARGER);

  return 0;
}
```

File: examples/Circular_kernel_3/functor_compare_theta_3.cpp

14.5 Design and Implementation History

This package follows the 2D circular kernel package (see Chapter 13), which induced the basic choices of design.

Julien Hazebrouck and Damien Leroy participated in a first prototype.

The first version of the package was co-authored by Pedro Machado Manhães de Castro and Monique Teillaud, and integrated in CGAL 3.4. Frédéric Cazals and Sébastien Loriot extended the package by providing functionalities restricted on a given sphere [dCCLT09].

Sylvain Pion is acknowledged for helpful discussions.

This work was partially supported by the IST Programme of the 6th Framework Programme of the EU as a STREP (FET Open Scheme) Project under Contract No IST-006413 (ACS - Algorithms for Complex Shapes).

3D Spherical Geometry Kernel Reference Manual

14.6 Geometric Concepts

Object types

Functors

14.7 Geometric Kernels and Classes

Kernels

Points

Arcs

Constants and Enumerations

14.8 Geometric Global Functions

14.9 Algebraic Concepts

Functors

14.10 Algebraic Kernel and Classes

Kernel

Polynomials

Roots of Polynomials

14.11 Alphabetical List of Reference Pages

SphericalKernel

Refines

Kernel

Has Models

CGAL::Spherical_kernel_3<Kernel,AlgebraicKernelForSpheres>
CGAL::Exact_spherical_kernel_3

Types

A model of *SphericalKernel* is supposed to provide some basic types

SphericalKernel:: Linear_kernel	Model of *Kernel*.
SphericalKernel:: Algebraic_kernel	Model of *AlgebraicKernelForSpheres*.

SphericalKernel:: Root_of_2	Model of *RootOf_2*.
SphericalKernel:: Root_for_spheres_2_3	Model of *AlgebraicKernelForSpheres::RootForSpheres_2_3*.
SphericalKernel:: Polynomial_1_3	Model of *AlgebraicKernelForSpheres::Polynomial_1_3*.
SphericalKernel:: Polynomials_for_line_3	Model of *AlgebraicKernelForSpheres::PolynomialsForLines_3*.
SphericalKernel:: Polynomial_for_spheres_2_3	Model of *AlgebraicKernelForSpheres::PolynomialForSpheres_2_3*.
SphericalKernel:: Polynomials_for_circle_3	Model of *AlgebraicKernelForSpheres::PolynomialsForCircles_3*.

and to define the following geometric objects

SphericalKernel:: Point_3	Model of *Kernel::Point_3*.
SphericalKernel:: Vector_3	Model of *Kernel::Vector_3*.
SphericalKernel:: Line_3	Model of *Kernel::Line_3*.
SphericalKernel:: Plane_3	Model of *Kernel::Plane_3*.
SphericalKernel:: Sphere_3	Model of *Kernel::Sphere_3*.
SphericalKernel:: Circle_3	Model of *Kernel::Circle_3*.

SphericalKernel:: Line_arc_3	Model of *SphericalKernel::LineArc_3*.
SphericalKernel:: Circular_arc_3	Model of *SphericalKernel::CircularArc_3*.
SphericalKernel:: Circular_arc_point_3	Model of *SphericalKernel::CircularArcPoint_3*.

Moreover, a model of *SphericalKernel* must provide predicates, constructions and other functionalities.

Predicates

SphericalKernel:: Compare_x_3	Model of *SphericalKernel::CompareX_3*.

SphericalKernel:: Compare_y_3	Model of *SphericalKernel::CompareY_3*.
SphericalKernel:: Compare_z_3	Model of *SphericalKernel::CompareZ_3*.
SphericalKernel:: Compare_xy_3	Model of *SphericalKernel::CompareXY_3*.
SphericalKernel:: Compare_xyz_3	Model of *SphericalKernel::CompareXYZ_3*.
SphericalKernel:: Compare_theta_3	Model of *SphericalKernel::CompareTheta_3*.
SphericalKernel:: Compare_theta_z_3	Model of *SphericalKernel::CompareThetaZ_3*.
SphericalKernel:: Compare_z_at_theta_3	Model of *SphericalKernel::CompareZAtTheta_3*.
SphericalKernel:: Compare_z_to_right_3	Model of *SphericalKernel::CompareZToRight_3*.
SphericalKernel:: Equal_3	Model of *SphericalKernel::Equal_3*.
SphericalKernel:: Has_on_3	Model of *SphericalKernel::HasOn_3*.
SphericalKernel:: Do_overlap_3	Model of *SphericalKernel::DoOverlap_3*.
SphericalKernel:: Do_intersect_3	Model of *SphericalKernel::DoIntersect_3*.
SphericalKernel:: Bounded_side_3	Model of *SphericalKernel::BoundedSide_3*.
SphericalKernel:: Has_on_bounded_side_3	Model of *SphericalKernel::HasOnBoundedSide_3*.
SphericalKernel:: Has_on_unbounded_side_3	Model of *SphericalKernel::HasOnUnboundedSide_3*.
SphericalKernel:: Is_theta_monotone_3	Model of *SphericalKernel::IsThetaMonotone_3*.

Constructions

SphericalKernel:: Construct_line_3	Model of *SphericalKernel::ConstructLine_3*.
SphericalKernel:: Construct_plane_3	Model of *SphericalKernel::ConstructPlane_3*.
SphericalKernel:: Construct_sphere_3	Model of *SphericalKernel::ConstructSphere_3*.
SphericalKernel:: Construct_circle_3	Model of *SphericalKernel::ConstructCircle_3*.
SphericalKernel:: Construct_line_arc_3	Model of *SphericalKernel::ConstructLineArc_3*.
SphericalKernel:: Construct_circular_arc_3	Model of *SphericalKernel::ConstructCircularArc_3*.
SphericalKernel:: Construct_circular_arc_point_3	Model of *SphericalKernel::ConstructCircularArcPoint_3*.
SphericalKernel:: Construct_circular_min_vertex_3	Model of *SphericalKernel::ConstructCircularMinVertex_3*.
SphericalKernel:: Construct_circular_max_vertex_3	Model of *SphericalKernel::ConstructCircularMaxVertex_3*.
SphericalKernel:: Construct_circular_source_vertex_3	Model of *SphericalKernel::ConstructCircularSourceVertex_3*.
SphericalKernel:: Construct_circular_target_vertex_3	Model of *SphericalKernel::ConstructCircularTargetVertex_3*.

SphericalKernel:: Construct_bbox_3	Model of *SphericalKernel::ConstructBbox_3*.
SphericalKernel:: Intersect_3	Model of *SphericalKernel::Intersect_3*.
SphericalKernel:: Split_3	Model of *SphericalKernel::Split_3*.
SphericalKernel:: Make_theta_monotone_3	Model of *SphericalKernel::MakeThetaMonotone_3*.

Computations

SphericalKernel:: Compute_circular_x_3	Model of *SphericalKernel::ComputeCircularX_3*.
SphericalKernel:: Compute_circular_y_3	Model of *SphericalKernel::ComputeCircularY_3*.
SphericalKernel:: Compute_circular_z_3	Model of *SphericalKernel::ComputeCircularZ_3*.

SphericalKernel:: Compute_approximate_squared_length_3

Model of *SphericalKernel::ComputeApproximateSquaredLength_3*.

SphericalKernel:: Compute_approximate_angle_3	Model of *SphericalKernel::ComputeApproximateAngle_3*.

Link with the algebraic kernel

SphericalKernel:: Get_equation	Model of *SphericalKernel::GetEquation*.

Operations

As in the *Kernel* concept, for each of the function objects above, there must exist a member function that requires no arguments and returns an instance of that function object. The name of the member function is the uncapitalized name of the type returned with the suffix *_object* appended. For example, for the function object *SphericalKernel::Construct_circular_arc_3* the following member function must exist:

Construct_circular_arc_3

sk.construct_circular_arc_3_object() const

For operations on a given sphere, a *context* sphere must be provided to the following functions:

Compare_theta_3	*sk.compare_theta_3_object(Sphere_3 sphere) const*
Compare_theta_z_3	*sk.compare_theta_z_3_object(Sphere_3 sphere) const*
Compare_z_at_theta_3	*sk.compare_z_at_theta_3_object(Sphere_3 sphere) const*
Compare_z_to_right_3	*sk.compare_z_to_right_3_object(Sphere_3 sphere) const*
Make_theta_monotone_3	*sk.make_theta_monotone_3_object(Sphere_3 sphere) const*

Is_theta_monotone_3　　*sk.is_theta_monotone_3_object(Sphere_3 sphere) const*

See Also

Kernel . . . . page 581

CGAL::Spherical_kernel_3<Kernel,AlgebraicKernelForSpheres>

Class

#include <CGAL/Spherical_kernel_3.h>

Is Model for the Concepts

SphericalKernel

Parameters

The first parameter of the spherical kernel must be instantiated with a model of the *Kernel* concept. The *Spherical_kernel_3* class template derives from this first parameter, in order to reuse all needed functionalities on basic objects provided by a model of the Kernel concept.

The second parameter, *AlgebraicKernelForSpheres*, is meant to provide the spherical kernel with all the algebraic functionalities required for the manipulation of algebraic curves.

Inherits From

Kernel

Types

The spherical kernel uses basic number types of the algebraic kernel:

typedef AlgebraicKernelForSpheres::RT	*RT;*	Ring number type.
typedef AlgebraicKernelForSpheres::FT	*FT;*	Field number type.

In fact, the two number types *AlgebraicKernelForSpheres::RT* and *Kernel::RT* must refer to the same type, as well as *AlgebraicKernelForSpheres::FT* and *Kernel::FT*.

The following types are available, as well as all the functionality on them described in the *SphericalKernel* concept.

typedef Line_arc_3<Spherical_kernel_3>	*Line_arc_3;*
typedef Circular_arc_3<Spherical_kernel_3>	*Circular_arc_3;*
typedef Circular_arc_point_3<Spherical_kernel_3>	*Circular_arc_point_3;*

Polynomials_for_circle_3 is implemented as a *std::pair< Polynomial_for_spheres_2_3, Polynomial_1_3 >* and is a model of *AlgebraicKernelForSpheres::PolynomialsForCircles_3*.

See Also

Kernel .... page 581
AlgebraicKernelForSpheres .... page 1322
CGAL::Exact_spherical_kernel_3 .... page 1260

Class

CGAL::Exact_spherical_kernel_3

#include <CGAL/Exact_spherical_kernel_3.h>

Definition

A typedef to a spherical kernel that provides both exact geometric predicates and exact geometric constructions.

It defines the same types as *CGAL::Spherical_kernel_3*.

Is Model for the Concepts

SphericalKernel

SphericalKernel::CircularArcPoint_3

Definition

Concept for points on spheres, circles, circular arcs or line arcs.

Refines

CopyConstructible, Assignable, DefaultConstructible

Has Models

CGAL::Circular_arc_point_3<SphericalKernel>

Class

CGAL::Circular_arc_point_3<SphericalKernel>

#include <CGAL/Circular_arc_point_3.h>

Is Model for the Concepts

SphericalKernel::CircularArcPoint_3

Creation

Circular_arc_point_3<SphericalKernel> p(Point_3<SphericalKernel> q);

Circular_arc_point_3<SphericalKernel> p(SphericalKernel::Root_for_spheres_2_3 r);

Access Functions

SphericalKernel::Root_of_2	*p.x()*	x-coordinate of the point.
SphericalKernel::Root_of_2	*p.y()*	y-coordinate of the point.
SphericalKernel::Root_of_2	*p.z()*	z-coordinate of the point.
Bbox_3	*p.bbox() const*	Returns a bounding box around the point.

Operations

bool	$p == q$	Test for equality. Two points are equal, iff their x, y and z coordinates are equal.
bool	$p\,!= q$	Test for nonequality.
bool	$p < q$	Returns true iff p is lexicographically smaller than q, i.e. either if $p.x() < q.x()$ or if $p.x() == q.x()$ and $p.y() < q.y()$ or if $p.x() == q.x()$ and $p.y() == q.y()$ and $p.z() < q.z()$.
bool	$p > q$	Returns true iff p is lexicographically greater than q.
bool	$p <= q$	Returns true iff p is lexicographically smaller than or equal to q.
bool	$p >= q$	Returns true iff p is lexicographically greater than or equal to q.

I/O

istream&	*std::istream& is >> Circular_arc_point_3 & p*

ostream& *std::ostream& os* << *Circular_arc_point_3 p*

See Also

CGAL::Circular_arc_3<SphericalKernel> . . . . page 1268
CGAL::Line_arc_3<SphericalKernel> . . . . page 1265

Concept

SphericalKernel::LineArc_3

Definition

Concept for line segments supported by a line that is a model of *Kernel::Line_3*, and whose endpoints are models of the *SphericalKernel::CircularArcPoint_3* concept.

Refines

CopyConstructible, Assignable, DefaultConstructible

Has Models

CGAL::Line_arc_3<SphericalKernel>

CGAL::Line_arc_3<SphericalKernel>

#include <CGAL/Line_arc_3.h>

Is Model for the Concepts

SphericalKernel::LineArc_3

Creation

Line_arc_3<SphericalKernel> la(Line_3<SphericalKernel> l,
Circular_arc_point_3<SphericalKernel> p1,
Circular_arc_point_3<SphericalKernel> p2)

Construct the line segment supported by *l*, whose source is *p1*, and whose target is *p2*.
Precondition: *p1* and *p2* lie on *l*. *p1* and *p2* are different.

Line_arc_3<SphericalKernel> la(Line_3<SphericalKernel> l,
Point_3<SphericalKernel> p1,
Point_3<SphericalKernel> p2)

Same.

Line_arc_3<SphericalKernel> la(Segment_3<SphericalKernel> s);

Access Functions

Line_3<SphericalKernel>	*la.supporting_line()*	
Circular_arc_point_3<SphericalKernel>	*la.source()*	
Circular_arc_point_3<SphericalKernel>	*la.target()*	
Circular_arc_point_3<SphericalKernel>	*la.min()*	Constructs the minimum vertex according to the lexicographic ordering of coordinates.
Circular_arc_point_3<SphericalKernel>	*la.max()*	Same for the maximum vertex.

Query Functions

bool	*la.is_vertical()*	Returns true *iff* the segment is vertical.

Operations

bool	*s1 == s2*	Test for equality. Two segments are equal, iff their non-oriented supporting lines are equal (i.e. they define the same set of points), and their endpoints are the same.
bool	*s1 != s2*	Test for nonequality.

I/O

istream&	*std::istream& is >> Line_arc_3 & ca*
ostream&	*std::ostream& os << Line_arc_3 ca*

The format for input/output is, for each line arc: a *Line_3* (the supporting line) and two *Circular_arc_point_3* (the two endpoints), under the condition that the endpoints are actually lying on the line.

See Also

CGAL::Circular_arc_point_3<SphericalKernel> .... page 1262
CGAL::Circular_arc_3<SphericalKernel> .... page 1268

SphericalKernel::CircularArc_3

Concept

Concept for arcs of circles.

Refines

CopyConstructible, Assignable, DefaultConstructible

Has Models

CGAL::Circular_arc_3<SphericalKernel>

Class

CGAL::Circular_arc_3<SphericalKernel>

#include <CGAL/Circular_arc_3.h>

Is Model for the Concepts

SphericalKernel::CircularArc_3

Creation

Circular_arc_3<SphericalKernel> ca(Circle_3<SphericalKernel> c);

Constructs an arc from a full circle.

Circular_arc_3<SphericalKernel> ca(Circle_3<SphericalKernel> c, Circular_arc_point_3 pt);

Constructs an arc from a full circle, using pt as source and target.

Circular_arc_3<SphericalKernel> ca(Circle_3<SphericalKernel> c,
Circular_arc_point_3<SphericalKernel> p,
Circular_arc_point_3<SphericalKernel> q)

Constructs the circular arc supported by c, whose source and target are p and q, respectively.
Precondition: p and q lie on c and are different.

The circular arc constructed from a circle, a source, and a target, is defined as the set of points of the circle that lie between the source *p1* and the target *p2*, when traversing the circle counterclockwise seen from the side of the plane of the circle pointed by its *positive* normal vectors.

In this definition, we say that a normal vector (a,b,c) is *positive* if $(a,b,c) > (0,0,0)$ (i.e. $(a > 0)||(a == 0)\&\&(b > 0)||(a == 0)\&\&(b == 0)\&\&(c > 0)$).

Circular_arc_3<SphericalKernel> ca(Point_3<SphericalKernel> p,
Point_3<SphericalKernel> q,
Point_3<SphericalKernel> r)

Constructs an arc that is supported by the circle of type *Circle_3<SphericalKernel>* passing through the points p, q and r. The source and target are respectively p and r, when traversing the supporting circle in the counterclockwise direction seen from the side of the plane containing the circle pointed by its *positive* normal vectors. Note that, depending on the orientation of the point triple *(p,q,r)*, q may not lie on the arc.
Precondition: p, q, and r are not collinear.

Access Functions

Circle_3<SphericalKernel>	*ca.supporting_circle()*	
Point_3<SphericalKernel>	*ca.center() const*	returns the center of the supporting circle.
SphericalKernel::FT	*ca.squared_radius() const*	returns the squared radius of the supporting circle.
Plane_3<SphericalKernel>	*ca.supporting_plane()*	
Sphere_3<SphericalKernel>	*ca.diametral_sphere()*	
Circular_arc_point_3<SphericalKernel>	*ca.source()*	
Circular_arc_point_3<SphericalKernel>	*ca.target()*	

When the methods *source* and *target* return the same point, then the arc is in fact a full circle.

When the arc was constructed from its (full) underlying circle, then source and target both return the smallest x-extremal point of the circle if the circle is not contained in a plane $x = A$, and the smallest y-extremal point otherwise.

Operations

bool	$a1 == a2$	Test for equality. Two arcs are equal, iff their non-oriented supporting planes are equal, and the centers and squared radii of their respective supporting circles are equal, and their sources and targets are equal.
bool	$a1 \,!= a2$	Test for nonequality.

I/O

istream&	*std::istream& is >> Circular_arc_3 & ca*
ostream&	*std::ostream& os << Circular_arc_3 ca*

The input/output format of a circular arc consists of the supporting circle represented as a *Circle_3* object followed by the source and target points of the arc represented as two *Circular_arc_point_3* objects. The defined arc is the unique arc constructed from such three objects.

See Also

CGAL::Circular_arc_point_3<SphericalKernel> . . . . page 1262
CGAL::Line_arc_3<SphericalKernel> . . . . page 1265

Enum

CGAL::Circle_type

Definition

The enum *Circle_type* is used to classify an object of type *Circle_3*, so as to specify its type (normal,polar,bipolar or threaded), as defined in section 14.2.

enum type { *NORMAL, THREADED, POLAR, BIPOLAR*}*;*

See Also

CGAL::classify . . . . page 1321

SphericalKernel::ConstructPlane_3

Refines

Kernel::ConstructPlane_3

A model *fo* of this concept must provide:

SphericalKernel::Plane_3

fo(SphericalKernel::Circular_arc_3 a)

Constructs the plane containing the arc.

SphericalKernel::Plane_3

fo(SphericalKernel::Polynomial_1_3 p)

Constructs a plane from an equation.

See Also

SphericalKernel::GetEquation . . . . page 1309

Functor
Concept

SphericalKernel::ConstructSphere_3

Refines

Kernel::ConstructSphere_3

A model *fo* of this concept must provide:

SphericalKernel::Sphere_3

fo(SphericalKernel::Circular_arc_3 a)

Returns the diametral sphere of the supporting circle of arc *a*.

SphericalKernel::Sphere_3

fo(SphericalKernel::Polynomial_2_3 p)

Constructs a sphere from an equation.

See Also

SphericalKernel::GetEquation . . . . page 1309

SphericalKernel::ConstructLine_3

Refines

Kernel::ConstructLine_3

A model *fo* of this concept must provide:

SphericalKernel::Line_3 *fo(SphericalKernel::Line_arc_3 s)*

Constructs the line containing the segment.

SphericalKernel::Line_3 *fo(SphericalKernel::Polynomials_for_lines_3 p)*

Constructs a line from an equation.

See Also

SphericalKernel::GetEquation . . . . page 1309

SphericalKernel::ConstructCircle_3

A model *fo* of this concept must provide:

SphericalKernel::Circle_3

fo(SphericalKernel::Circular_arc_3 a)

Constructs the circle containing the arc.

SphericalKernel::Circle_3

fo(SphericalKernel::Polynomials_for_circles_3 p)

Constructs a circle from an equation.

See Also

SphericalKernel::GetEquation . . . . page 1309

SphericalKernel::ConstructCircularArcPoint_3

A model *fo* of this concept must provide:

SphericalKernel::Circular_arc_point_3

fo(SphericalKernel::Root_for_spheres_2_3 r)

SphericalKernel::Circular_arc_point_3

fo(SphericalKernel::Point_3 p)

SphericalKernel::ConstructLineArc_3

A model *fo* of this concept must provide:

SphericalKernel::Line_arc_3

*fo.operator()(SphericalKernel::Line_3 l,
SphericalKernel::Circular_arc_point_3 p,
SphericalKernel::Circular_arc_point_3 q)*

Constructs the line segment supported by *l*, whose source is *p* and whose target is *q*.
Precondition: *p* and *q* lie on *l* and are different.

SphericalKernel::Line_arc_3

fo(SphericalKernel::Segment_3 s)

SphericalKernel::Line_arc_3

fo(SphericalKernel::Point_3 p, SphericalKernel::Point_3 q)

SphericalKernel::ConstructCircularArc_3

A model *fo* of this concept must provide:

SphericalKernel::Circular_arc_3

fo(SphericalKernel::Circle_3 c)

Constructs an arc from a full circle.

SphericalKernel::Circular_arc_3

fo.operator()(SphericalKernel::Circle_3 c,
SphericalKernel::Circular_arc_point_3 p,
SphericalKernel::Circular_arc_point_3 q)

Constructs the circular arc supported by c, whose source and target are p and q, respectively.
Precondition: p and q lie on c and they are different.

The circular arc constructed from a circle, a source, and a target, is defined as the set of points of the circle that lie between the source *p1* and the target *p2*, when traversing the circle counterclockwise seen from the side of the plane of the circle pointed by its *positive* normal vectors.

In this definition, we say that a normal vector (a,b,c) is *positive* if $(a,b,c) > (0,0,0)$ (i.e. $(a > 0)||(a == 0)\&\&(b > 0)||(a == 0)\&\&(b == 0)\&\&(c > 0)$).

SphericalKernel::Circular_arc_3

fo.operator()(SphericalKernel::Point_3 p,
SphericalKernel::Point_3 q,
SphericalKernel::Point_3 r)

Constructs an arc that is supported by the circle of type *SphericalKernel::Circle_3* passing through the points p, q and r. The source and target are respectively p and r, when traversing the supporting circle in the counterclockwise direction seen from the side of the plane containing the circle pointed by its *positive* normal vectors. the circle. Note that, depending on the orientation of the point triple *(p,q,r)*, q may not lie on the arc.
Precondition: p, q, and r are not collinear.

SphericalKernel::ConstructCircularMinVertex_3

A model *fo* of this concept must provide:

SphericalKernel::Circular_arc_point_3

fo(SphericalKernel::Line_arc_3 l)

Constructs the minimal vertex of *l* with lexicographically smallest coordinates.

SphericalKernel::ConstructCircularMaxVertex_3

A model *fo* of this concept must provide:

SphericalKernel::Circular_arc_point_3

fo(SphericalKernel::Line_arc_3 l)

Constructs the maximal vertex of *l* with lexicographically largest coordinates.

Functor
Concept

SphericalKernel::ConstructCircularSourceVertex_3

A model *fo* of this concept must provide:

SphericalKernel::Circular_arc_point_3

fo(SphericalKernel::Circular_arc_3 a)

Constructs the source vertex of *a*.

SphericalKernel::Circular_arc_point_3

fo(SphericalKernel::Line_arc_3 l)

Same, for a line segment.

SphericalKernel::ConstructCircularTargetVertex_3

A model *fo* of this concept must provide:

SphericalKernel::Circular_arc_point_3

fo(SphericalKernel::Circular_arc_3 a)

Constructs the target vertex of *a*.

SphericalKernel::Circular_arc_point_3

fo(SphericalKernel::Line_arc_3 l)

Same, for a line segment.

SphericalKernel::ConstructBbox_3

A model *fo* of this concept must provide operators to construct a bounding box of geometric objects:

CGAL::Bbox_3	*fo(SphericalKernel::Circular_arc_point_3 p)*
CGAL::Bbox_3	*fo(SphericalKernel::Line_arc_3 l)*
CGAL::Bbox_3	*fo(SphericalKernel::Circular_arc_3 a)*

SphericalKernel::Split_3

A model *fo* of this type must provide:

void *fo.operator()(SphericalKernel::Circular_arc_3 a, SphericalKernel::Circular_arc_point_3 p, SphericalKernel::Circular_arc_3 &a1, SphericalKernel::Circular_arc_3 &a2)*

Splits arc a at point p, which creates arcs $a1$ and $a2$.
Precondition: The point p lies in the interior of the input arc a.

void *fo.operator()(SphericalKernel::Line_arc_3 l, SphericalKernel::Circular_arc_point_3 p, SphericalKernel::Line_arc_3 &l1, SphericalKernel::Line_arc_3 &l2)*

Same for a line arc.

Functor
Concept

SphericalKernel::MakeThetaMonotone_3

A model *fo* of this concept must provide:

template<class OutputIterator>
OutputIterator *fo(SphericalKernel::Circular_arc_3 a, OutputIterator res)*

Copies in the output iterator the results of the split of arc *a* at the θ-extremal point(s) of its supporting circle relatively to the context sphere used by the function *SphericalKernel::make_theta_monotone_3_object* (Refer to section 14.2 for the definition of these points.) The output iterator may contain no arc (if the supporting circle is a bipolar circle), one arc (if *a* is already θ-monotone), two arcs (if only one θ-extremal point is on *a*), or three arcs (if two θ-extremal points are on *a*).
Precondition: *a* lies on the context sphere used by the function *SphericalKernel::make_theta_monotone_3_object*, and the supporting circle of *a* is not bipolar.

template<class OutputIterator>
OutputIterator *fo(SphericalKernel::Circle_3 c, OutputIterator res)*

Copies in the output iterator the results of the split of circle *c* at its θ-extremal point(s) relatively to the context sphere used by the function *SphericalKernel::make_theta_monotone_3_object*. (Refer to section 14.2 for the definition of these points.) The output iterator may contain no arc (if the circle is bipolar), one arc (if the circle is polar or threaded), or two arcs (if the circle is normal). The source and target are such that the circular arc is the set of points of the circle that lie between the source and the target when traversing the circle counterclockwise seen from the positive side of the plane of the circle. In this definition, we say that a normal vector (a,b,c) is *positive* if $(a > 0)||(a == 0)\&\&(b > 0)||(a == 0)\&\&(b == 0)\&\&(c > 0)$.
For a threaded circle, the arc returned the one built using the full circle.
For a polar circle, the arc returned is the full circle, the source and target correspond to the pole the circle goes through.
Precondition: *c* lies on the context sphere used by the function *SphericalKernel::make_theta_monotone_3_object*, and *c* is not bipolar.

See Also

SphericalKernel::CompareX_3

Functor
Concept

Refines

Kernel::CompareX_3

An object *fo* of this type must provide in addition:

Comparison_result *fo.operator()(SphericalKernel::Circular_arc_point_3 p, SphericalKernel::Circular_arc_point_3 q)*

Compares the x-coordinates of p and q.

See Also

SphericalKernel::CompareY_3

Refines

Kernel::CompareY_3

An object *fo* of this type must provide in addition:

Comparison_result *fo.operator()(SphericalKernel::Circular_arc_point_3 p, SphericalKernel::Circular_arc_point_3 q)*

Compares the y-coordinates of p and q.

See Also

SphericalKernel::CompareZ_3

Functor
Concept

Refines

Kernel::CompareZ_3

An object *fo* of this type must provide in addition:

Comparison_result *fo.operator()(SphericalKernel::Circular_arc_point_3 p, SphericalKernel::Circular_arc_point_3 q)*

Compares the z-coordinates of p and q.

See Also

Functor
Concept

SphericalKernel::CompareXY_3

Refines

Kernel::CompareXY_3

An object *fo* of this type must provide in addition:

Comparison_result *fo.operator()(SphericalKernel::Circular_arc_point_3 p, SphericalKernel::Circular_arc_point_3 q)*

Compares p and q according to the lexicographic ordering on x- and y-coordinates.

See Also

SphericalKernel::CompareXYZ_3

Functor
Concept

Refines

Kernel::CompareXYZ_3

An object *fo* of this type must provide in addition:

Comparison_result *fo.operator()(SphericalKernel::Circular_arc_point_3 p, SphericalKernel::Circular_arc_point_3 q)*

Compares p and q according to the lexicographic ordering on x-, y-, and z-coordinates.

See Also

SphericalKernel::CompareX_3 . . . . . page 1285
SphericalKernel::CompareY_3 . . . . . page 1286
SphericalKernel::CompareZ_3 . . . . . page 1287
SphericalKernel::CompareXY_3 . . . . . page 1288
SphericalKernel::CompareTheta_3 . . . . . page 1290
SphericalKernel::CompareThetaZ_3 . . . . . page 1291
SphericalKernel::Equal_3 . . . . . page 1294

Functor
Concept

SphericalKernel::CompareTheta_3

An object *fo* of this type must provide:

Comparison_result *fo.operator()(SphericalKernel::Circular_arc_point_3 p, SphericalKernel::Circular_arc_point_3 q)*

Compares the θ-coordinates of p and q in the cylindrical coordinate system relative to the context sphere used by the function *SphericalKernel::compare_theta_3_object*.
Precondition: p and q lie on the context sphere used by the function *SphericalKernel::compare_theta_3_object*, but do not coincide with its poles.

Comparison_result *fo(SphericalKernel::Circular_arc_point_3 p, SphericalKernel::Vector_3 m)*

Compares the θ-coordinates of p and of the meridian defined by m (see section 14.2) in the cylindrical coordinate system relative to the context sphere used by the function *SphericalKernel::compare_theta_3_object*.
Precondition: p lies on the context sphere used by the function *SphericalKernel::compare_theta_3_object*, but does not coincide with its poles. $m \neq (0,0,0)$ and the z-coordinate of m is 0.

Comparison_result *fo(SphericalKernel::Vector_3 m, SphericalKernel::Circular_arc_point_3 p)*

Same as previous, with opposite result.

Comparison_result *fo(SphericalKernel::Vector_3 m1, SphericalKernel::Vector_3 m2)*

Compares the θ-coordinates of the meridians defined by $m1$ and by $m2$ (see section 14.2) in the cylindrical coordinate system relative to the context sphere used by the function *SphericalKernel::compare_theta_3_object*.
$m1 \neq (0,0,0)$, $m2 \neq (0,0,0)$ and the z-coordinate of $m1$ and $m2$ is 0.

See Also

SphericalKernel::CompareThetaZ_3 . . . . . page 1291
SphericalKernel::CompareX_3 . . . . . page 1285
SphericalKernel::CompareY_3 . . . . . page 1286
SphericalKernel::CompareZ_3 . . . . . page 1287
SphericalKernel::CompareXY_3 . . . . . page 1288
SphericalKernel::CompareXYZ_3 . . . . . page 1289
SphericalKernel::Equal_3 . . . . . page 1294

SphericalKernel::CompareThetaZ_3

An object *fo* of this type must provide:

Comparison_result *fo.operator()(SphericalKernel::Circular_arc_point_3 p, SphericalKernel::Circular_arc_point_3 q)*

Compares p and q according to the lexicographic ordering on θ- and z-coordinates in the cylindrical coordinate system relative to the context sphere used by the function *SphericalKernel::compare_theta_z_3_object*.
Precondition: p and q lie on the context sphere used by the function *SphericalKernel::compare_theta_z_3_object*, but do not coincide with its poles.

See Also

SphericalKernel::CompareTheta_3 . . . page 1290
SphericalKernel::CompareX_3 . . . page 1285
SphericalKernel::CompareY_3 . . . page 1286
SphericalKernel::CompareZ_3 . . . page 1287
SphericalKernel::CompareXY_3 . . . page 1288
SphericalKernel::CompareXYZ_3 . . . page 1289
SphericalKernel::Equal_3 . . . page 1294

Functor
Concept

SphericalKernel::CompareZAtTheta_3

An object *fo* of this type must provide:

Comparison_result *fo.operator()(SphericalKernel::Circular_arc_3 a0, SphericalKernel::Circular_arc_3 a1, SphericalKernel::Vector_3 m)*

compares the z-coordinates of the two intersections points of $a0$ and $a1$ with the meridian defined by m (see section 14.2).
Precondition: *a0* and *a1* lie on the context sphere used by the function *SphericalKernel::compare_z_at_theta_3_object*. $m \neq (0,0,0)$ and the z-coordinate of m is 0. Arcs $a0$ and $a1$ are θ-monotone and both intersected by the meridian defined by m (see section 14.2).

Comparison_result *fo(SphericalKernel::Circular_arc_point_3 p, SphericalKernel::Circular_arc_3 a)*

given a meridian anchored at the poles of the context sphere used by the function *SphericalKernel::compare_z_at_theta_3_object*, and passing through point p, compares the z-coordinate of point p and that of the intersection of the meridian with a.
Precondition: a and p lie on the context sphere used by the function *SphericalKernel::compare_z_at_theta_3_object*, arc a is θ-monotone and the meridian passing through p intersects arc a.

See Also

SphericalKernel::CompareZToRight_3 . . . . page 1293

SphericalKernel::CompareZToRight_3

An object *fo* of this type must provide:

Comparison_result *fo.operator()(SphericalKernel::Circular_arc_3 a0, SphericalKernel::Circular_arc_3 a1, SphericalKernel::Circular_arc_point_3 p)*

Compares the z-coordinates of the intersection points of both arcs with a meridian anchored at the poles of the context sphere used by the function *SphericalKernel::compare_z_to_right_3_object*, at a θ-coordinate infinitesimally greater that the θ-coordinate of point p.
Precondition: *a0* and *a1* lie on the context sphere used by the function *SphericalKernel::compare_z_to_right_3_object*, *a0* and *a1* are θ-monotone, p lies on *a0* and *a1* and is not a θ-extremal point of the supporting circle of *a0* or *a1*.

See Also

SphericalKernel::CompareZAtTheta_3 . . . . page 1292

SphericalKernel::Equal_3

Definition

Testing equality between objects.

Refines

Kernel::Equal_3

An object *fo* of this type must provide in addition:

bool *fo.operator()(SphericalKernel::Circular_arc_point_3 p0, SphericalKernel::Circular_arc_point_3 p1)*

For two points.

bool *fo(SphericalKernel::Circular_arc_3 a0, SphericalKernel::Circular_arc_3 a1)*

For two arcs. Two arcs are equal, iff their non-oriented supporting planes are equal, and the centers and squared radii of their respective supporting circles are equal, and their sources and targets are equal.

bool *fo(SphericalKernel::Line_arc_3 a0, SphericalKernel::Line_arc_3 a1)*

For two segments. Two segments are equal, iff their non-oriented supporting lines are equal (i.e. they define the same set of points), and their endpoints are the same.

See Also

SphericalKernel::CompareX_3 . . . . page 1285
SphericalKernel::CompareY_3 . . . . page 1286
SphericalKernel::CompareZ_3 . . . . page 1287
SphericalKernel::CompareXY_3 . . . . page 1288
SphericalKernel::CompareXYZ_3 . . . . page 1289

SphericalKernel::HasOn_3

Refines

Kernel::HasOn_3

An object *fo* of this type must provide:

bool *fo(Type1 obj1, Type2 obj2)*

Returns *true* when *obj1* contains *obj2*,

where *Type1* and *Type2* can be respectively:

Type1	*Type2*
SphericalKernel::Sphere_3, *SphericalKernel::Plane_3*, *SphericalKernel::Line_3*, *SphericalKernel::Circle_3*, *SphericalKernel::Line_arc_3*, or *SphericalKernel::Circular_arc_3*	*SphericalKernel::Circular_arc_point_3*
SphericalKernel::Plane_3, *SphericalKernel::Line_3*	*SphericalKernel::Line_arc_3*
SphericalKernel::Sphere_3, *SphericalKernel::Circle_3*, or *SphericalKernel::Plane_3*	*SphericalKernel::Circular_arc_3*

SphericalKernel::DoOverlap_3

Refines

Kernel::DoOverlap_3

An object *fo* of this type must provide:

bool	*fo.operator()(*	*SphericalKernel::Line_arc_3 l0, SphericalKernel::Line_arc_3 l1, const bool known_equal_supporting_line = false)*
		For two line arcs.
bool	*fo.operator()(*	*SphericalKernel::Circular_arc_3 a0, SphericalKernel::Circular_arc_3 a1, const bool known_equal_supporting_circle = false)*
		For two circular arcs.

The computation may be faster when the boolean is set to *true*.

SphericalKernel::DoIntersect_3

Definition

Testing whether two curves intersect.

Refines

Kernel::DoIntersect_3

An object *fo* of this type must provide:

bool	*fo(Type1 obj1, Type2 obj2)*	
		determines if two geometric objects of type Type1 and Type2 intersect or not.

for all pairs *Type1* and *Type2*, where the types *Type1* and *Type2* can be either, any of the following:

- *SphericalKernel::Plane_3*
- *SphericalKernel::Line_3*
- *SphericalKernel::Line_arc_3*
- *SphericalKernel::Sphere_3*
- *SphericalKernel::Circle_3*

See Also

CGAL::do_intersect . . . . page 730

Functor Concept

SphericalKernel::BoundedSide_3

Refines

Kernel::BoundedSide_3

An object *fo* of this type must provide:

Bounded_side	*fo(SphericalKernel::Sphere_3 s, SphericalKernel::Circular_arc_point_3 p)*
	For a sphere.
Bounded_side	*fo(SphericalKernel::Circle_3 s, SphericalKernel::Circular_arc_point_3 p)*
	For a circle.

See Also

SphericalKernel::HasOnBoundedSide_3 . . . . page 1299
SphericalKernel::HasOnUnboundedSide_3 . . . . page 1300

SphericalKernel::HasOnBoundedSide_3

Functor
Concept

Refines

Kernel::HasOnBoundedSide_3

An object *fo* of this type must provide:

bool	*fo(SphericalKernel::Sphere_3 s, SphericalKernel::Circular_arc_point_3 p)*
	For a sphere.
bool	*fo(SphericalKernel::Circle_3 s, SphericalKernel::Circular_arc_point_3 p)*
	For a circle.

See Also

SphericalKernel::BoundedSide_3 . . . . page 1298
SphericalKernel::HasOnUnboundedSide_3 . . . . page 1300

SphericalKernel::HasOnUnboundedSide_3

Refines

Kernel::HasOnUnboundedSide_3

An object *fo* of this type must provide:

bool	*fo(SphericalKernel::Sphere_3 s, SphericalKernel::Circular_arc_point_3 p)*
	For a sphere.
bool	*fo(SphericalKernel::Circle_3 s, SphericalKernel::Circular_arc_point_3 p)*
	For a circle.

See Also

SphericalKernel::BoundedSide_3 . . . . page 1298
SphericalKernel::HasOnBoundedSide_3 . . . . page 1299

SphericalKernel::IsThetaMonotone_3

An object *fo* of this type must provide:

bool *fo(SphericalKernel::Circular_arc_3 a)*

Tests whether the arc *a* is θ-monotone, i.e. the intersection of any meridian anchored at the poles of the context sphere used by the function *SphericalKernel::is_theta_monotone_3_object* and the arc *a* is reduced to at most one point in general, and two points if a pole of that sphere is an endpoint of *a*. Note that a bipolar circle has no such arcs.
Precondition: *a* lies on the context sphere used by the function *SphericalKernel::is_theta_monotone_3_object*, and the supporting circle of *a* is not bipolar.

Functor Concept

See Also

SphericalKernel::MakeThetaMonotone_3 . . . . page 1284

Functor
Concept

SphericalKernel::Intersect_3

Refines

Kernel::Intersect_3

A model *fo* of this concept must provide:

template < class OutputIterator >
OutputIterator *fo(Type1 obj1, Type2 obj2, OutputIterator intersections)*

Copies in the output iterator the intersection elements between the two objects. *intersections* iterates on elements of type *CGAL::Object*, in lexicographic order when this ordering is defined on the computed objects.

template < class OutputIterator >
OutputIterator *fo(Type1 obj1, Type2 obj2, Type3 obj3, OutputIterator intersections)*

Copies in the output iterator the intersection elements between the three objects. *intersections* iterates on elements of type *CGAL::Object*, in lexicographic order when this ordering is defined on the computed objects.

For the **first operator**, *Type1* and *Type2* can both be either

- *SphericalKernel::Sphere_3*,
- *SphericalKernel::Plane_3*,
- *SphericalKernel::Line_3*,
- *SphericalKernel::Circle_3*,
- *SphericalKernel::Line_arc_3* or
- *SphericalKernel::Circular_arc_3*,

depending on the types *Type1* and *Type2*, the computed *CGAL::Object*s can be assigned to

- *std::pair<SphericalKernel::Circular_arc_point_3, unsigned>*, where the unsigned integer is the multiplicity of the corresponding intersection point between *obj_1* and *obj_2*,
- *Type1*, when *Type1* and *Type2* are equal, and if the two objects *obj1* and *obj2* are equal,
- *SphericalKernel::Line_3* or *SphericalKernel::Circle_3* when *Type1* and *Type2* are two-dimensional objects intersecting along a curve (2 planes, or 2 spheres, or one plane and one sphere),
- *SphericalKernel::Circular_arc_3* in case of an overlap of two circular arcs or
- *SphericalKernel::Line_arc_3* in case of an overlap of two line segments.

For the **second operator**, *Type1*, *Type2* and *Type3* can be either

- *SphericalKernel::Sphere_3* or
- *SphericalKernel::Plane_3*

and depending of these types, the computed *CGAL::Object*s can be assigned to

- *std::pair<SphericalKernel::Circular_arc_point_3, unsigned>*, where the unsigned integer is the multiplicity of the corresponding intersection point,
- *SphericalKernel::Circle_3* or
- *Type1*, when *Type1*, *Type2* and *Type3* are equal, and if the three objects *obj1* and *obj2* and *obj3* are equal.

SphericalKernel::ComputeCircularX_3

SphericalKernel::Root_of_2

fo(SphericalKernel::Circular_arc_point_3 p)

returns the x-coordinate of the point p.

SphericalKernel::ComputeCircularY_3

SphericalKernel::Root_of_2

fo(SphericalKernel::Circular_arc_point_3 p)

returns the y-coordinate of the point p.

SphericalKernel::ComputeCircularZ_3

SphericalKernel::Root_of_2

fo(SphericalKernel::Circular_arc_point_3 p)

returns the z-coordinate of the point p.

SphericalKernel::ComputeApproximateSquaredLength_3

Refines

Kernel::ComputeApproximateSquaredLength_3

In addition, an object *fo* of this type must provide:

double	*fo(SphericalKernel::Circular_arc_3 a)*	
		Computes an approximation of the squared length of the arc *a*.

Functor
Concept

SphericalKernel::ComputeApproximateAngle_3

An object *fo* of this type must provide:

double	*fo(SphericalKernel::Circular_arc_3 a)*	
		Computes an approximation of the angle of the arc in radians *a*.

SphericalKernel::GetEquation

A model *fo* of this concept must provide:

SphericalKernel::Polynomial_1_3

fo(SphericalKernel::Plane_3 p)

Returns the equation of the plane.

SphericalKernel::Polynomial_for_spheres_2_3

fo(SphericalKernel::Sphere_3 p)

Returns the equation of the sphere.

SphericalKernel::Polynomials_for_line_3

fo(SphericalKernel::Line_3 c)

Returns the equations of the line.

SphericalKernel::Polynomials_for_circle_3

fo(SphericalKernel::Circle_3 c)

Returns the equations of the circle.

See Also

SphericalKernel::ConstructPlane_3 . . . . page 1271
SphericalKernel::ConstructSphere_3 . . . . page 1272
SphericalKernel::ConstructLine_3 . . . . page 1273
SphericalKernel::ConstructCircle_3 . . . . page 1274

Function

CGAL::is_theta_monotone

#include <CGAL/global_functions_spherical_kernel_3.h>

bool *is_theta_monotone(Circular_arc_3<SphericalKernel> a, Sphere_3<SphericalKernel> sphere)*

Tests whether the arc *a* is θ-monotone, i.e. the intersection of any meridian anchored at the poles *sphere* and the arc *a* is reduced to at most one point in general, and two points if a pole of *sphere* is an endpoint of the arc. Note that a bipolar circle has no such arcs.
Precondition: *a* lies on *sphere*, and the supporting circle of *a* is not bipolar.

CGAL::compare_theta

#include <CGAL/global_functions_spherical_kernel_3.h>

Comparison_result *compare_theta(Circular_arc_point_3<SphericalKernel> p, Circular_arc_point_3<SphericalKernel> q, Sphere_3<SphericalKernel> sphere)*

Compares the θ-coordinates of p and q relatively to *sphere*.
Precondition: p and q lie on *sphere*, but do not coincide with the poles of *sphere*.

Comparison_result *compare_theta(SphericalKernel::Circular_arc_point_3 p, SphericalKernel::Vector_3 m, SphericalKernel::Sphere_3 sphere)*

Compares the θ-coordinates of p and of the meridian defined by m (see section 14.2) in the cylindrical coordinate system relative to *sphere* .
Precondition: p lies on *sphere*, but does not coincide with its poles. $m \neq (0,0,0)$ and the z-coordinate of m is 0.

Comparison_result *compare_theta(SphericalKernel::Vector_3 m, SphericalKernel::Circular_arc_point_3 p)*

Same as previous, with opposite result.

See Also

CGAL::compare_x . . . . . . page 712
CGAL::compare_xy . . . . . . page 714
CGAL::compare_xyz . . . . . . page 716
CGAL::compare_x_at_y . . . . . . page 717
CGAL::compare_y . . . . . . page 719
CGAL::compare_yx . . . . . . page 723
CGAL::compare_y_at_x . . . . . . page 721
CGAL::compare_z . . . . . . page 724
CGAL::compare_theta_z . . . . . . page 1312

Function

CGAL::compare_theta_z

#include <CGAL/global_functions_spherical_kernel_3.h>

bool *compare_theta_z(Circular_arc_point_3<SphericalKernel> p, Circular_arc_point_3<SphericalKernel> q, Sphere_3<SphericalKernel> sphere)*

Compares p and q according to the lexicographic ordering on θ and z-coordinates in the cylindrical coordinate system relative to *sphere*.
Precondition: p and q lie on *sphere*, but do not coincide with the poles of *sphere*.

See Also

CGAL::compare_x . . . . . page 712
CGAL::compare_xy . . . . . page 714
CGAL::compare_xyz . . . . . page 716
CGAL::compare_x_at_y . . . . . page 717
CGAL::compare_y . . . . . page 719
CGAL::compare_yx . . . . . page 723
CGAL::compare_y_at_x . . . . . page 721
CGAL::compare_z . . . . . page 724
CGAL::compare_theta . . . . . page 1311

CGAL::x_extremal_point

#include <CGAL/global_functions_spherical_kernel_3.h>

Circular_arc_point_3<SphericalKernel>

x_extremal_point(Sphere_3<SphericalKernel> c, bool b)

Returns the point on the sphere that is extremal in the x-direction, and that is the smallest (resp. largest) of the two x-extremal points for the lexicographic order if b is *true* (resp. *false*).

Circular_arc_point_3<SphericalKernel>

x_extremal_point(Circle_3<SphericalKernel> c, bool b)

Same for a circle.
Precondition: The circle is not contained in a plane orthogonal to the x-axis.

Function

Function

CGAL::y_extremal_point

#include <CGAL/global_functions_spherical_kernel_3.h>

Circular_arc_point_3<SphericalKernel>

y_extremal_point(Sphere_3<SphericalKernel> c, bool b)

Returns the point on the sphere that is extremal in the y-direction, and that is the smallest (resp. largest) of the two y-extremal points for the lexicographic order if b is *true* (resp. *false*).

Circular_arc_point_3<SphericalKernel>

y_extremal_point(Circle_3<SphericalKernel> c, bool b)

Same for a circle.
Precondition: The circle is not contained in a plane orthogonal to the y-axis.

CGAL::z_extremal_point

#include <CGAL/global_functions_spherical_kernel_3.h>

Circular_arc_point_3<SphericalKernel>

z_extremal_point(Sphere_3<SphericalKernel> c, bool b)

Returns the point on the sphere that is extremal in the z-direction, and that is the smallest (resp. largest) of the two z-extremal points for the lexicographic order if b is *true* (resp. *false*).

Circular_arc_point_3<SphericalKernel>

z_extremal_point(Circle_3<SphericalKernel> c, bool b)

Same for a circle.
Precondition: The circle is not contained in a plane orthogonal to the z-axis.

Function

Function

CGAL::theta_extremal_point

#include <CGAL/global_functions_spherical_kernel_3.h>

Circular_arc_point_3<SphericalKernel>

theta_extremal_point(Circle_3<SphericalKernel> c,
const Sphere_3<SphericalKernel> sphere,
bool b)

Returns the point on the circle that is extremal in θ using the cylindrical coordinate system relative to *sphere*, and that has the smallest (resp. largest) θ-coordinate of the two points if *b* is *true* (resp. *false*). See section 14.2 for definitions.
Precondition: *c* lies on *sphere* and is a normal circle.

CGAL::x_extremal_points

#include <CGAL/global_functions_spherical_kernel_3.h>

template < class OutputIterator >
OutputIterator *x_extremal_points(Sphere_3<SphericalKernel> c, OutputIterator res)*

Copies in the output iterator the x-extremal points of the sphere. *res* iterates on elements of type *Circular_arc_point_3<SphericalKernel>*, lexicographically sorted.

template < class OutputIterator >
OutputIterator *x_extremal_points(Circle_3<SphericalKernel> c, OutputIterator res)*

Copies in the output iterator the x-extremal points of the circle. *res* iterates on elements of type *Circular_arc_point_3<SphericalKernel>*, lexicographically sorted.
Precondition: The circle is not contained in a plane orthogonal to the x-axis.

Function

Function

CGAL::y_extremal_points

#include <CGAL/global_functions_spherical_kernel_3.h>

template < class OutputIterator >
OutputIterator *y_extremal_points(Sphere_3<SphericalKernel> c, OutputIterator res)*

Copies in the output iterator the y-extremal points of the sphere. *res* iterates on elements of type *Circular_arc_point_3<SphericalKernel>*, lexicographically sorted.

template < class OutputIterator >
OutputIterator *y_extremal_points(Circle_3<SphericalKernel> c, OutputIterator res)*

Copies in the output iterator the y-extremal points of the circle. *res* iterates on elements of type *Circular_arc_point_3<SphericalKernel>*, lexicographically sorted.
Precondition: The circle is not contained in a plane orthogonal to the y-axis.

CGAL::z_extremal_points

#include <CGAL/global_functions_spherical_kernel_3.h>

template < class OutputIterator >
OutputIterator *z_extremal_points(Sphere_3<SphericalKernel> c, OutputIterator res)*

Copies in the output iterator the z-extremal points of the sphere. *res* iterates on elements of type *Circular_arc_point_3<SphericalKernel>*, lexicographically sorted.

template < class OutputIterator >
OutputIterator *z_extremal_points(Circle_3<SphericalKernel> c, OutputIterator res)*

Copies in the output iterator the z-extremal points of the circle. *res* iterates on elements of type *Circular_arc_point_3<SphericalKernel>*, lexicographically sorted.
Precondition: The circle is not contained in a plane orthogonal to the z-axis.

Function

CGAL::theta_extremal_points

#include <CGAL/global_functions_spherical_kernel_3.h>

template < class OutputIterator >
OutputIterator *theta_extremal_points(Circle_3<SphericalKernel> c, Sphere_3<SphericalKernel> sphere, OutputIterator res)*

Copies in the output iterator the θ-extremal points of the circle relatively to *sphere*. *res* iterates on elements of type *Circular_arc_point_3<SphericalKernel>*, lexicographically sorted in the cylindrical coordinate system relative to *sphere*. See section 14.2 for definitions.
Precondition: *c* lies on *sphere* and is a normal circle.

CGAL::classify

#include <CGAL/global_functions_spherical_kernel_3.h>

CGAL::Circle_type	*classify(Circle_3<SphericalKernel> c, Sphere_3<SphericalKernel> sphere)*	
		Classify a circle according to *sphere*, as defined in section 14.2. *Precondition*: *c* lies on *sphere*.

Function

See Also

CGAL::Circle_type . . . . page 1270

AlgebraicKernelForSpheres

Definition

The *AlgebraicKernelForSpheres* concept is meant to provide the curved kernel with all the algebraic functionalities required for the manipulation of spheres, circles, and circular arcs in 3D.

Has Models

Algebraic_kernel_for_spheres_2_3

Types

A model of *AlgebraicKernelForSpheres* is supposed to provide

AlgebraicKernelForSpheres:: RT	A model of *RingNumberType*.
AlgebraicKernelForSpheres:: FT	A model of *FieldNumberType<RT>*.
AlgebraicKernelForSpheres:: Polynomial_1_3	A model of *AlgebraicKernelForSpheres::Polynomial_1_3*, for trivariate polynomials of degree up to 1.
AlgebraicKernelForSpheres:: Polynomial_for_spheres_2_3	A model of *AlgebraicKernelForSpheres::PolynomialForSpheres_2_3*, for trivariate polynomials of degree up to 2 that can store equations of spheres.
AlgebraicKernelForSpheres:: Polynomials_for_lines_3	A model of *AlgebraicKernelForSpheres::PolynomialsForLines_3*, for systems of polynomials that can store equations of lines in 3D.
AlgebraicKernelForSpheres:: Root_of_2	A model of *RootOf_2*, for algebraic numbers of degree up to 2.
AlgebraicKernelForSpheres:: Root_for_spheres_2_3	A model of *AlgebraicKernelForSpheres::RootForSpheres_2_3*, for solutions of systems of three models of *AlgebraicKernelForSpheres::PolynomialForSpheres_2_3*.
AlgebraicKernelForSpheres:: Construct_polynomial_1_3	A model of *AlgebraicKernelForSpheres::ConstructPolynomial_1_3*.

AlgebraicKernelForSpheres:: Construct_polynomial_for_spheres_2_3

A model of *AlgebraicKernelForSpheres::ConstructPolynomialForSpheres_2_3*.

AlgebraicKernelForSpheres:: Construct_polynomials_for_lines_3

A model of *AlgebraicKernelForSpheres::ConstructPolynomialsForLines_3*.

AlgebraicKernelForSpheres:: Compare_x	A model of the concept *AlgebraicKernelForSpheres::CompareX*.
AlgebraicKernelForSpheres:: Compare_y	A model of the concept *AlgebraicKernelForSpheres::CompareY*.
AlgebraicKernelForSpheres:: Compare_z	A model of the concept *AlgebraicKernelForSpheres::CompareZ*.
AlgebraicKernelForSpheres:: Compare_xy	A model of the concept *AlgebraicKernelForSpheres::CompareXY*.
AlgebraicKernelForSpheres:: Compare_xyz	A model of the concept *AlgebraicKernelForSpheres::CompareXYZ*.
AlgebraicKernelForSpheres:: Sign_at	A model of the concept *AlgebraicKernelForSpheres::SignAt*.
AlgebraicKernelForSpheres:: X_critical_points	A model of the concept *AlgebraicKernelForSpheres::XCriticalPoints*.
AlgebraicKernelForSpheres:: Y_critical_points	A model of the concept *AlgebraicKernelForSpheres::YCriticalPoints*.
AlgebraicKernelForSpheres:: Z_critical_points	A model of the concept *AlgebraicKernelForSpheres::ZCriticalPoints*.
AlgebraicKernelForSpheres:: Solve	A model of the concept *AlgebraicKernelForSpheres::Solve*.

See Also

SphericalKernel . . . . page 1255
CGAL::Spherical_kernel_3<Kernel,AlgebraicKernelForSpheres> . . . . page 1259

Class

CGAL::Algebraic_kernel_for_spheres_2_3<RT>

#include <CGAL/Algebraic_kernel_for_spheres_2_3.h>

Is Model for the Concepts

AlgebraicKernelForSpheres

AlgebraicKernelForSpheres::Polynomial_1_3

Concept

Definition

Concept to represent trivariate polynomials of degree 1 whose coefficients are of a type that is a model of the concept *RingNumberType*.

Refines

CopyConstructible, Assignable, DefaultConstructible

Has Models

CGAL::Polynomial_1_3

See Also

AlgebraicKernelForSpheres . . . . page 1322

Class

CGAL::Polynomial_1_3<RT>

#include <CGAL/Polynomials_1_3.h>

Is Model for the Concepts

AlgebraicKernelForSpheres::Polynomial_1_3

Concept

AlgebraicKernelForSpheres::PolynomialForSpheres_2_3

Definition

Concept to represent trivariate polynomials of degree up to 2 capable of storing equations of spheres, whose center's coordinates, as well as the square of the radius, are of a type that is a model of the concept *FieldNumberType*.

Refines

CopyConstructible, Assignable, DefaultConstructible

Creation

Operations

The comparison operator == must be provided.

bool	$p == q$	Tests equality of two polynomials.

Has Models

CGAL::Polynomial_for_spheres_2_3

See Also

AlgebraicKernelForSpheres . . . . page 1322

Class

CGAL::Polynomial_for_spheres_2_3<FT>

#include <CGAL/Polynomials_2_3.h>

Is Model for the Concepts

AlgebraicKernelForSpheres::PolynomialForSpheres_2_3

See Also

AlgebraicKernelForSpheres . . . . page 1322

AlgebraicKernelForSpheres::PolynomialsForLines_3

Definition

Concept to represent a system of polynomials on *FieldNumberType* capable of storing equations of lines.

Refines

CopyConstructible, Assignable, DefaultConstructible

Has Models

CGAL::Polynomials_for_lines_3

See Also

AlgebraicKernelForSpheres . . . . page 1322

Class

CGAL::Polynomials_for_lines_3<FT>

#include <CGAL/Polynomials_for_line_3.h>

Is Model for the Concepts

AlgebraicKernelForSpheres::PolynomialsForLines_3

See Also

AlgebraicKernelForSpheres . . . . page 1322

AlgebraicKernelForSpheres::PolynomialsForCircles_3

Concept

Definition

Concept to represent a system of polynomials on *FieldNumberType* capable of storing equations of circles.

Refines

CopyConstructible, Assignable, DefaultConstructible

See Also

AlgebraicKernelForSpheres . . . . page 1322

Concept

AlgebraicKernelForSpheres::RootForSpheres_2_3

Definition

Concept to represent the roots of a system of three equations of degree 2 in three variables x, y and z that are models of concept *AlgebraicKernelForSpheres::PolynomialForSpheres_2_3*.

Operations

The comparison operator == must be provided.

bool	*r1 == r2*	Test equality of two roots of systems.

Has Models

CGAL::Root_for_spheres_2_3

See Also

AlgebraicKernelForSpheres . . . . page 1322

CGAL::Root_for_spheres_2_3<RT>

#include <CGAL/Root_for_spheres_2_3.h>

Is Model for the Concepts

AlgebraicKernelForSpheres::RootForSpheres_2_3

Class

AlgebraicKernelForSpheres::CompareX

A model *fo* of this concept must provide:

CGAL::Comparison_result

fo.operator()(AlgebraicKernelForSpheres::Root_for_spheres_2_3 r1, AlgebraicKernelForSpheres::Root_for_spheres_2_3 r2)

Compares the x (first) variables of two *Root_for_spheres_2_3*.

See Also

AlgebraicKernelForSpheres::CompareY

Definition

A model *fo* of this concept must provide:

CGAL::Comparison_result

fo.operator()(AlgebraicKernelForSpheres::Root_for_spheres_2_3 r1, AlgebraicKernelForSpheres::Root_for_spheres_2_3 r2)

Compares the y (second) variables of two *Root_for_spheres_2_3*.

See Also

AlgebraicKernelForSpheres::CompareX . . . . . page 1334
AlgebraicKernelForSpheres::CompareZ . . . . . page 1336
AlgebraicKernelForSpheres::CompareXY . . . . . page 1337
AlgebraicKernelForSpheres::CompareXYZ . . . . . page 1338

Concept

AlgebraicKernelForSpheres::CompareZ

Definition

A model *fo* of this concept must provide:

CGAL::Comparison_result

fo.operator()(AlgebraicKernelForSpheres::Root_for_spheres_2_3 r1, AlgebraicKernelForSpheres::Root_for_spheres_2_3 r2)

Compares the z (third) variables of two *Root_for_spheres_2_3*.

See Also

AlgebraicKernelForSpheres::CompareX . . . . page 1334
AlgebraicKernelForSpheres::CompareY . . . . page 1335
AlgebraicKernelForSpheres::CompareXY . . . . page 1337
AlgebraicKernelForSpheres::CompareXYZ . . . . page 1338

AlgebraicKernelForSpheres::CompareXY

Definition

A model *fo* of this concept must provide:

CGAL::Comparison_result

fo.operator()(AlgebraicKernelForSpheres::Root_for_spheres_2_3 r1, AlgebraicKernelForSpheres::Root_for_spheres_2_3 r2)

Compares the x and y (the two first) variables of two *Root_for_spheres_2_3* lexicographically.

See Also

Concept

AlgebraicKernelForSpheres::CompareXYZ

Definition

A model *fo* of this concept must provide:

CGAL::Comparison_result

fo.operator()(AlgebraicKernelForSpheres::Root_for_spheres_2_3 r1, AlgebraicKernelForSpheres::Root_for_spheres_2_3 r2)

Compares two *Root_for_spheres_2_3* lexicographically.

See Also

AlgebraicKernelForSpheres::CompareX . . . . page 1334
AlgebraicKernelForSpheres::CompareY . . . . page 1335
AlgebraicKernelForSpheres::CompareZ . . . . page 1336
AlgebraicKernelForSpheres::CompareXY . . . . page 1337

AlgebraicKernelForSpheres::SignAt

Definition

A model *fo* of this concept must provide:

CGAL::Sign	*fo.operator()(AlgebraicKernelForSpheres::Polynomial_1_3 p, AlgebraicKernelForSpheres::Root_for_spheres_2_3 r)*	
		Computes the sign of polynomial *p* evaluated at a root *r*.
CGAL::Sign	*fo.operator()(AlgebraicKernelForSpheres::Polynomial_for_spheres_2_3 p, AlgebraicKernelForSpheres::Root_for_spheres_2_3 r)*	
		Same as previous.

Concept

AlgebraicKernelForSpheres::ConstructPolynomial_1_3

A model *fo* of this concept must provide:

AlgebraicKernelForSpheres::Polynomial_1_3

fo.operator()(AlgebraicKernelForSpheres::RT a,
AlgebraicKernelForSpheres::RT b,
AlgebraicKernelForSpheres::RT c,
AlgebraicKernelForSpheres::RT d)

Constructs polynomial $ax+by+cz+d$.

See Also

SphericalKernel::ConstructPlane_3 . . . . page 1271
SphericalKernel::GetEquation . . . . page 1309

AlgebraicKernelForSpheres::ConstructPolynomialForSpheres_2_3

Concept

A model *fo* of this concept must provide:

AlgebraicKernelForSpheres::PolynomialForSpheres_2_3

fo.operator()(AlgebraicKernelForSpheres::FT a,
AlgebraicKernelForSpheres::FT b,
AlgebraicKernelForSpheres::FT c,
AlgebraicKernelForSpheres::FT rsq)

Constructs polynomial *(x-a)*ˆ*2 + (y-b)*ˆ*2 + (z-c)*ˆ*2 - rsq*.

See Also

SphericalKernel::ConstructSphere_3 . . . . page 1272
SphericalKernel::GetEquation . . . . page 1309

Concept

AlgebraicKernelForSpheres::ConstructPolynomialsForLines_3

A model *fo* of this concept must provide:

AlgebraicKernelForSpheres::Polynomials_for_line_3

*fo.operator()(AlgebraicKernelForSpheres::FT a1,
AlgebraicKernelForSpheres::FT a2,
AlgebraicKernelForSpheres::FT b1,
AlgebraicKernelForSpheres::FT b2,
AlgebraicKernelForSpheres::FT c1,
AlgebraicKernelForSpheres::FT c2)*

Constructs polynomials *a1.t+a2*, *b1.t+b2*, *c1.t+c2*.

See Also

SphericalKernel::ConstructLine_3 . . . . page 1273
SphericalKernel::GetEquation . . . . page 1309

AlgebraicKernelForSpheres::Solve

Definition

A model *fo* of this concept must provide:

template < class OutputIterator >
OutputIterator *fo(Type1 p1, Type2 p2, Type3 p3, OutputIterator res)*

Copies in the output iterator the common roots of *p1*, *p2*, and *p3*, with their multiplicity, as objects of type *std::pair< AlgebraicKernelForSpheres::Root_for_spheres_2_3, int>*.
Precondition: The set of solutions of the system is 0-dimensional.

Here, *Type1*, *Type2*, and *Type3* can all be either *AlgebraicKernelForSpheres::Polynomial_1_3* or *AlgebraicKernelForSpheres::Polynomial_for_spheres_2_3*.

Concept

AlgebraicKernelForSpheres::XCriticalPoints

Definition

A model *fo* of this concept must provide:

template < class OutputIterator >
OutputIterator *fo.operator()(AlgebraicKernelForSpheres::Polynomial_for_spheres_2_3 p, OutputIterator res)*

Copies in the output iterator the x-critical points of polynomial p, as objects of type *AlgebraicKernelForSpheres::Root_for_spheres_2_3*.

template < class OutputIterator >
AlgebraicKernelForSpheres::Root_for_spheres_2_3
fo(AlgebraicKernelForSpheres::Polynomial_for_spheres_2_3 p, bool i)

Computes the ith x-critical point of polynomial p.

See Also

AlgebraicKernelForSpheres::YCriticalPoints . . . . page 1345
AlgebraicKernelForSpheres::ZCriticalPoints . . . . page 1346

AlgebraicKernelForSpheres::YCriticalPoints

Definition

A model *fo* of this concept must provide:

template < class OutputIterator >
OutputIterator *fo.operator()(AlgebraicKernelForSpheres::Polynomial_for_spheres_2_3 p, OutputIterator res)*

Copies in the output iterator the y-critical points of polynomial p, as objects of type *AlgebraicKernelForSpheres::Root_for_spheres_2_3*.

template < class OutputIterator >
AlgebraicKernelForSpheres::Root_for_spheres_2_3
fo(AlgebraicKernelForSpheres::Polynomial_for_spheres_2_3 p, bool i)

Computes the ith y-critical point of polynomial p.

See Also

AlgebraicKernelForSpheres::XCriticalPoints . . . . page 1344
AlgebraicKernelForSpheres::ZCriticalPoints . . . . page 1346

AlgebraicKernelForSpheres::ZCriticalPoints

Definition

A model *fo* of this concept must provide:

template < class OutputIterator >
OutputIterator *fo.operator()(AlgebraicKernelForSpheres::Polynomial_for_spheres_2_3 p, OutputIterator res)*

Copies in the output iterator the *z*-critical points of polynomial *p*, as objects of type *AlgebraicKernelForSpheres::Root_for_spheres_2_3*.

template < class OutputIterator >
AlgebraicKernelForSpheres::Root_for_spheres_2_3

fo(AlgebraicKernelForSpheres::Polynomial_for_spheres_2_3 p, bool i)

Computes the *i*th *z*-critical point of polynomial *p*.

See Also

AlgebraicKernelForSpheres::XCriticalPoints . . . . page 1344
AlgebraicKernelForSpheres::YCriticalPoints . . . . page 1345

Part V

Convex Hull Algorithms

Chapter 15

2D Convex Hulls and Extreme Points

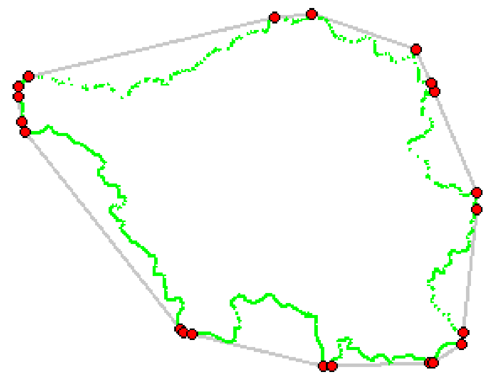
Susan Hert and Stefan Schirra

Contents

15.1 Introduction

A subset $S \subseteq \mathbb{R}^2$ is convex if for any two points p and q in the set the line segment with endpoints p and q is contained in S. The convex hull of a set S is the smallest convex set containing S. The convex hull of a set of points P is a convex polygon with vertices in P. A point in P is an extreme point (with respect to P) if it is a vertex of the convex hull of P. A set of points is said to be strongly convex if it consists of only extreme points.

This chapter describes the functions provided in CGAL for producing convex hulls in two dimensions as well as functions for checking if sets of points are strongly convex are not. There are also a number of functions described for computing particular extreme points and subsequences of hull points, such as the lower and upper hull of a set of points.



15.2 Convex Hull

CGAL provides implementations of several classical algorithms for computing the counterclockwise sequence of extreme points for a set of points in two dimensions (*i.e.*, the counterclockwise sequence of points on the convex hull). The algorithms have different asymptotic running times and require slightly different sets of geometric primitives. Thus you may choose the algorithm that best fits your setting.

Each of the convex hull functions presents the same interface to the user. That is, the user provides a pair of iterators, *first* and *beyond*, an output iterator *result*, and a traits class *traits*. The points in the range [*first*, *beyond*) define the input points whose convex hull is to be computed. The counterclockwise sequence of extreme points is written to the sequence starting at position *result*, and the past-the-end iterator for the resulting set of points is returned. The traits classes for the functions specify the types of the input points and the geometric primitives that are required by the algorithms. All functions provide an interface in which this class need not be specified and defaults to types and operations defined in the kernel in which the input point type is defined.

Given a sequence of n input points with h extreme points, the function *convex_hull_2* uses either the output-sensitive $O(nh)$ algorithm of Bykat [Byk78] (a non-recursive version of the quickhull [BDH96] algorithm) or the algorithm of Akl and Toussaint, which requires $O(n \log n)$ time in the worst case. The algorithm chosen depends on the kind of iterator used to specify the input points. These two algorithms are also available via the functions *ch_bykat* and *ch_akl_toussaint*, respectively. Also available are the $O(n \log n)$ Graham-Andrew scan algorithm [And79, Meh84] (*ch_graham_andrew*), the $O(nh)$ Jarvis march algorithm [Jar73] (*ch_jarvis*), and Eddy's $O(nh)$ algorithm [Edd77] (*ch_eddy*), which corresponds to the two-dimensional version of the quickhull algorithm. The linear-time algorithm of Melkman for producing the convex hull of simple polygonal chains (or polygons) is available through the function *ch_melkman*.

15.3 Example using Graham-Andrew's Algorithm

In the following example a convex hull is constructed from point data read from standard input using *Graham_Andrew* algorithm. The resulting convex polygon is shown at the standard output console. The same results could be achieved by substituting the function *CGAL::ch_graham_andrew* by other function like *CGAL::ch_bykat*.

```
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/ch_graham_andrew.h>
```

```
typedef CGAL::Exact_predicates_inexact_constructions_kernel K;
typedef K::Point_2 Point_2;

int main()
{
  CGAL::set_ascii_mode(std::cin);
  CGAL::set_ascii_mode(std::cout);
  std::istream_iterator< Point_2 >  in_start( std::cin );
  std::istream_iterator< Point_2 >  in_end;
  std::ostream_iterator< Point_2 >  out( std::cout, "\n" );
  CGAL::ch_graham_andrew( in_start, in_end, out );
  return 0;
}
```

File: examples/Convex_hull_2/ch_from_cin_to_cout.cpp

15.4 Extreme Points and Hull Subsequences

In addition to the functions for producing convex hulls, there are a number of functions for computing sets and sequences of points related to the convex hull. The functions *lower_hull_points_2* and *upper_hull_points_2* provide the computation of the counterclockwise sequence of extreme points on the lower hull and upper hull, respectively. The algorithm used in these functions is Andrew's variant of Graham's scan algorithm [And79, Meh84], which has worst-case running time of $O(n \log n)$.

There are also functions available for computing certain subsequences of the sequence of extreme points on the convex hull. The function *ch_jarvis_march* generates the counterclockwise ordered subsequence of extreme points between a given pair of points and *ch_graham_andrew_scan* computes the sorted sequence of extreme points that are not left of the line defined by the first and last input points.

Finally, a set of functions (*ch_nswe_point*, *ch_ns_point*, *ch_we_point*, *ch_n_point*, *ch_s_point*, *ch_w_point*, *ch_e_point*) is provided for computing extreme points of a 2D point set in the coordinate directions.

15.5 Traits Classes

Each of the functions used to compute convex hulls or extreme points is parameterized by a traits class, which specifies the types and geometric primitives to be used in the computation. There are several implementations of 2D traits classes provided in the library. The class *Convex_hull_traits_2<R>* corresponds to the default traits class that provides the types and predicates presented in the 2-dimensional CGAL kernel in which the input points lie. The class *Convex_hull_constructive_traits<R>* is a second traits class based on CGAL primitives but differs from *Convex_hull_traits_2* in that some of its primitives reuse intermediate results to speed up computation.

In addition, the 2D and 3D Linear Geometric Kernel provides three projective traits classes (*Projection_traits_xy_3<K>*, *Projection_traits_zx_3<K*, and *Projection_traits_yz_3<K*),

which may be used to compute the convex hull of a set of three-dimensional points projected into each of the three coordinate planes.

15.6 Convexity Checking

The functions *is_ccw_strongly_convex_2* and *is_cw_strongly_convex_2* check whether a given sequence of 2D points forms a (counter)clockwise strongly convex polygon.. These are used in postcondition testing of the two-dimensional convex hull functions.

2D Convex Hulls and Extreme Points Reference Manual

Susan Hert and Stefan Schirra

A subset $S \subseteq \mathbb{R}^2$ is convex if for any two points p and q in the set the line segment with endpoints p and q is contained in S. The convex hull of a set S is the smallest convex set containing S. The convex hull of a set of points P is a convex polygon with vertices in P. A point in P is an extreme point (with respect to P) if it is a vertex of the convex hull of P.

CGAL provides functions for computing convex hulls in two dimensions as well as functions for testing if a given set of points is strongly convex or not. There are also a number of functions available for computing particular extreme points in 2D and subsequences of the hull points, such as the lower hull or upper hull of a set of points.

15.7 Classified Reference Pages

Assertions

The assertion flags for the convex hull and extreme point algorithms use *CH* in their names (*e.g.*, *CGAL_CH_NO_POSTCONDITIONS*). For the convex hull algorithms, the postcondition check tests only convexity (if not disabled), but not containment of the input points in the polygon or polyhedron defined by the output points. The latter is considered an expensive checking and can be enabled by defining *CGAL_CH_CHECK_EXPENSIVE*.

Concepts

Traits Classes

Convex Hull Functions

Convexity Checking Functions

Hull Subsequence Functions

Extreme Point Functions

15.8 Alphabetical List of Reference Pages

Function

CGAL::ch_akl_toussaint

Definition

The function *ch_akl_toussaint* generates the counterclockwise sequence of extreme points from a given set of input points.

#include <CGAL/ch_akl_toussaint.h>

```
template <class ForwardIterator, class OutputIterator, class Traits>
OutputIterator    ch_akl_toussaint( ForwardIterator first,
                                    ForwardIterator beyond,
                                    OutputIterator result,
                                    Traits ch_traits = Default_traits())
```

generates the counterclockwise sequence of extreme points of the points in the range [*first*,*beyond*). The resulting sequence is placed starting at position *result*, and the past-the-end iterator for the resulting sequence is returned. It is not specified at which point the cyclic sequence of extreme points is cut into a linear sequence.
Precondition: The source range [*first*,*beyond*) does not contain *result*.

The default traits class *Default_traits* is the kernel in which the type *ForwardIterator::value_type* is defined.

Requirements

1. *ForwardIterator::value_type* and *OutputIterator::value_type* are equivalent to *Traits::Point_2*.
2. *Traits* defines the following subset of types from the concept *ConvexHullTraits_2* and their corresponding member functions that return instances of these types:
   - *Traits::Point_2*,
   - *Traits::Less_xy_2*,
   - *Traits::Less_yx_2*,
   - *Traits::Left_turn_2*,
   - *Traits::Equal_2*.

See Also

CGAL::ch_bykat . . . . page 1358
CGAL::ch_eddy . . . . page 1360
CGAL::ch_graham_andrew . . . . page 1363
CGAL::ch_jarvis . . . . page 1367
CGAL::ch_melkman . . . . page 1371
CGAL::convex_hull_2 . . . . page 1379

Implementation

This function uses the algorithm of Akl and Toussaint [AT78] that requires $O(n \log n)$ time for n input points.

Function

CGAL::ch_bykat

Definition

The function *ch_bykat* generates the counterclockwise sequence of extreme points from a given set of input points.

#include <CGAL/ch_bykat.h>

template <class InputIterator, class OutputIterator, class Traits>
OutputIterator ch_bykat(InputIterator first, InputIterator beyond, OutputIterator result, Traits ch_traits = Default_traits)

generates the counterclockwise sequence of extreme points of the points in the range [*first*,*beyond*). The resulting sequence is placed starting at position *result*, and the past-the-end iterator for the resulting sequence is returned. It is not specified at which point the cyclic sequence of extreme points is cut into a linear sequence.
Precondition: The source range [*first*,*beyond*) does not contain *result*.

The default traits class *Default_traits* is the kernel in which the type *ForwardIterator::value_type* is defined.

Requirements

1. *InputIterator::value_type* and *OutputIterator::value_type* are equivalent to *Traits::Point_2*.
2. *Traits* defines the following subset of types from the concept *ConvexHullTraits_2* and their corresponding member functions that return instances of these types:
   - *Traits::Point_2*,
   - *Traits::Less_signed_distance_to_line_2*,
   - *Traits::Left_turn_2*,
   - *Traits::Less_xy_2*,
   - *Traits::Equal_2*.

See Also

CGAL::ch_akl_toussaint . . . . . page 1356
CGAL::ch_eddy . . . . . page 1360
CGAL::ch_graham_andrew . . . . . page 1363
CGAL::ch_jarvis . . . . . page 1367
CGAL::ch_melkman . . . . . page 1371
CGAL::convex_hull_2 . . . . . page 1379

Implementation

This function implements the non-recursive variation of Eddy's algorithm [Edd77] described in [Byk78]. This algorithm requires $O(nh)$ time in the worst case for n input points with h extreme points.

CGAL::ch_eddy

Definition

The function *ch_eddy* generates the counterclockwise sequence of extreme points from a given set of input points.

#include <CGAL/ch_eddy.h>

template <class InputIterator, class OutputIterator, class Traits>
OutputIterator ch_eddy(InputIterator first, InputIterator beyond, OutputIterator result, Traits ch_traits = Default_traits)

generates the counterclockwise sequence of extreme points of the points in the range [*first*,*beyond*). The resulting sequence is placed starting at position *result*, and the past-the-end iterator for the resulting sequence is returned. It is not specified at which point the cyclic sequence of extreme points is cut into a linear sequence.
Precondition: The source range [*first*,*beyond*) does not contain *result*.

The default traits class *Default_traits* is the kernel in which the type *ForwardIterator::value_type* is defined.

Requirements

1. *InputIterator::value_type* and *OutputIterator::value_type* are equivalent to *Traits::Point_2*.
2. *Traits* defines the following subset of types from the concept *ConvexHullTraits_2* and their corresponding member functions that return instances of these types:
   - *Traits::Point_2*,
   - *Traits::Equal_2*,
   - *Traits::Less_signed_distance_to_line_2*,
   - *Traits::Left_turn_2*,
   - *Traits::Less_xy_2*.

See Also

CGAL::ch_akl_toussaint . . . . page 1356
CGAL::ch_bykat . . . . page 1358
CGAL::ch_graham_andrew . . . . page 1363
CGAL::ch_jarvis . . . . page 1367
CGAL::ch_melkman . . . . page 1371
CGAL::convex_hull_2 . . . . page 1379

Implementation

This function implements Eddy's algorithm [Edd77], which is the two-dimensional version of the quickhull algorithm [BDH96] . This algorithm requires $O(nh)$ time in the worst case for n input points with h extreme points.

Function

CGAL::ch_e_point

Definition

The function *ch_e_point* finds a point of a given set of input points with maximal x coordinate.

#include <CGAL/ch_selected_extreme_points_2.h>

template <class ForwardIterator>
void *ch_e_point(ForwardIterator first, ForwardIterator beyond, ForwardIterator& e, Traits ch_traits = Default_traits)*

traverses the range [*first*,*beyond*). After execution, the value of e is an iterator in the range such that $*e \geq_{xy} *it$ for all iterators *it* in the range.

The default traits class *Default_traits* is the kernel in which the type *ForwardIterator::value_type* is defined.

Requirements

Traits defines a type *Traits::Less_xy_2* as described in the concept *ConvexHullTraits_2* and the corresponding member function that returns an instance of this type.

See Also

CGAL::ch_nswe_point . . . . page 1373
CGAL::ch_n_point . . . . page 1375
CGAL::ch_ns_point . . . . page 1374
CGAL::ch_s_point . . . . page 1376
CGAL::ch_w_point . . . . page 1378
CGAL::ch_we_point . . . . page 1377

CGAL::ch_graham_andrew

Function

Definition

The function *ch_graham_andrew* generates the counterclockwise sequence of extreme points from a given set of input points.

#include <CGAL/ch_graham_andrew.h>

template <class InputIterator, class OutputIterator, class Traits>
OutputIterator ch_graham_andrew(InputIterator first, InputIterator beyond, OutputIterator result, Traits ch_traits = Default_traits)

generates the counterclockwise sequence of extreme points of the points in the range [*first*,*beyond*). The resulting sequence is placed starting at position *result*, and the past-the-end iterator for the resulting sequence is returned. It is not specified at which point the cyclic sequence of extreme points is cut into a linear sequence.
Precondition: The source range [*first*,*beyond*) does not contain *result*.

The default traits class *Default_traits* is the kernel in which the type *InputIterator::value_type* is defined.

Requirements

1. *InputIterator::value_type* and *OutputIterator::value_type* are equivalent to *Traits::Point_2*.
2. *Traits* defines the following subset of types from the concept *ConvexHullTraits_2* and their corresponding member functions that return instances of these types:
   - *Traits::Point_2*,
   - *Traits::Less_xy_2*,
   - *Traits::Left_turn_2*,
   - *Traits::Equal_2*.

See Also

CGAL::ch_akl_toussaint . . . . . . page 1356
CGAL::ch_bykat . . . . . . page 1358
CGAL::ch_eddy . . . . . . page 1360
CGAL::ch_graham_andrew_scan . . . . . . page 1365
CGAL::ch_jarvis . . . . . . page 1367
CGAL::ch_melkman . . . . . . page 1371
CGAL::convex_hull_2 . . . . . . page 1379

Implementation

This function implements Andrew's variant of the Graham scan algorithm [And79] and follows the presentation of Mehlhorn [Meh84]. This algorithm requires $O(n \log n)$ time in the worst case for n input points.

CGAL::ch_graham_andrew_scan

Function

Definition

The function *ch_graham_andrew_scan* generates the counterclockwise sequence of extreme points from a given set of input points that are not left of the line defined by the first and last points in this sequence.

#include <CGAL/ch_graham_andrew.h>

template <class BidirectionalIterator, class OutputIterator, class Traits>
OutputIterator ch_graham_andrew_scan(BidirectionalIterator first, BidirectionalIterator beyond, OutputIterator result, Traits ch_traits = Default_traits)

generates the counterclockwise sequence of extreme points that are not left of pq, where p is the value of *first* and q is the value of *beyond* -1. The resulting sequence is placed starting at *result* with p; point q is omitted. The past-the-end iterator for the sequence is returned.
Precondition: The range [*first*,*beyond*) contains at least two different points. The points in [*first*,*beyond*) are "sorted" with respect to pq, *i.e.*, the sequence of points in [*first*,*beyond*) define a counterclockwise polygon, for which the Graham-Sklansky-procedure [Skl72] works.

The default traits class *Default_traits* is the kernel in which the type *BidirectionalIterator::value_type* is defined.

Requirements

1. *BidirectionalIterator::value_type* and *OutputIterator::value_type* are equivalent to *Traits::Point_2*.
2. *Traits* defines the following two types from the concept *ConvexHullTraits_2* and their corresponding member functions that return instances of these types:
   - *Traits::Point_2*,
   - *Traits::Left_turn_2*.

See Also

CGAL::ch_graham_andrew . . . . page 1363
CGAL::lower_hull_points_2 . . . . page 1393
CGAL::upper_hull_points_2 . . . . page 1395

Implementation

The function uses Andrew's variant of the Graham scan algorithm [And79] . This algorithm requires $O(n \log n)$ time in the worst case for n input points.

Example

In the following example *ch_graham_andrew_scan()* is used to realize Anderson's variant [And78] of the Graham Scan [Gra72]. The points are sorted counterclockwise around the leftmost point using the *Less_rotate_ccw_2* predicate, as defined in the concept *ConvexHullTraits_2*. According to the definition of *Less_rotate_ccw_2*, the leftmost point is the last point in the sorted sequence and its predecessor on the convex hull is the first point in the sorted sequence. It is not hard to see that the preconditions of *ch_graham_andrew_scan()* are satisfied. Anderson's variant of the Graham scan is usually inferior to Andrew's variant because of its higher arithmetic demand.

```
template <class InputIterator, class OutputIterator, class Traits>
OutputIterator
ch_graham_anderson( InputIterator  first, InputIterator  beyond,
                    OutputIterator result, const Traits&  ch_traits)
{
  typedef typename Traits::Less_xy_2          Less_xy_2;
  typedef typename Traits::Point_2            Point_2;
  typedef typename Traits::Less_rotate_ccw_2  Less_rotate_ccw_2;

  if (first == beyond) return result;
  std::vector< Point_2 >  V;
  copy( first, beyond, back_inserter(V) );
  typename std::vector< Point_2 >::iterator it =
              std::min_element(V.begin(), V.end(), Less_xy_2());
  std::sort( V.begin(), V.end(), CGAL::bind_1(Less_rotate_ccw_2(), *it) );
  if ( *(V.begin()) == *(V.rbegin()) )
  {
      *result = *(V.begin());  ++result;
      return result;
  }
  return ch_graham_andrew_scan( V.begin(), V.end(), result, ch_traits);
}
```

CGAL::ch_jarvis

Function

Definition

The function *ch_jarvis* generates the counterclockwise sequence of extreme points from a given set of input points.

#include <CGAL/ch_jarvis.h>

template <class InputIterator, class OutputIterator, class Traits>
OutputIterator ch_jarvis(InputIterator first, InputIterator beyond, OutputIterator result, Traits ch_traits = Default_traits)

generates the counterclockwise sequence of extreme points of the points in the range [*first*,*beyond*). The resulting sequence is placed starting at position *result*, and the past-the-end iterator for the resulting sequence is returned. It is not specified at which point the cyclic sequence of extreme points is cut into a linear sequence.
Precondition: The source range [*first*,*beyond*) does not contain *result*.

The default traits class *Default_traits* is the kernel in which the type *InputIterator::value_type* is defined.

Requirements

1. *InputIterator::value_type* and *OutputIterator::value_type* are equivalent to *Traits::Point_2*.
2. *Traits* defines the following subset of types from the concept *ConvexHullTraits_2* and their corresponding member functions that return instances of these types:
   - *Traits::Point_2*,
   - *Traits::Equal_2*,
   - *Traits::Less_rotate_ccw_2*,
   - *Traits::Less_xy_2*.

See Also

CGAL::ch_akl_toussaint . . . . page 1356
CGAL::ch_bykat . . . . page 1358
CGAL::ch_eddy . . . . page 1360
CGAL::ch_graham_andrew . . . . page 1363
CGAL::ch_jarvis_march . . . . page 1369
CGAL::ch_melkman . . . . page 1371
CGAL::convex_hull_2 . . . . page 1379

Implementation

This function uses the Jarvis march (gift-wrapping) algorithm [Jar73]. This algorithm requires $O(nh)$ time in the worst case for n input points with h extreme points.

CGAL::ch_jarvis_march

Function

Definition

The function *ch_jarvis_march* generates the counterclockwise sequence of extreme points from a given set of input points that line between two input points.

#include <CGAL/ch_jarvis.h>

template <class ForwardIterator, class OutputIterator, class Traits>
OutputIterator ch_jarvis_march(ForwardIterator first, ForwardIterator beyond, Traits::Point_2 start_p, Traits::Point_2 stop_p, OutputIterator result, Traits ch_traits = Default_traits)

generates the counterclockwise subsequence of extreme points between *start_p* and *stop_p* of the points in the range [*first*,*beyond*), starting at position *result* with point *start_p*. The last point generated is the point preceding *stop_p* in the counterclockwise order of extreme points.

Precondition: *start_p* and *stop_p* are extreme points with respect to the points in the range [*first*,*beyond*) and *stop_p* is an element of range [*first*,*beyond*).

The default traits class *Default_traits* is the kernel in which the type *ForwardIterator::value_type* is defined.

Requirements

1. *ForwardIterator::value_type* and *OutputIterator::value_type* are equivalent to *Traits::Point_2*.
2. *Traits* defines the following subset of types from the concept *ConvexHullTraits_2* and their corresponding member functions that return instances of these types:
   - *Traits::Point_2*,
   - *Traits::Equal_2*,
   - *Traits::Less_rotate_ccw_2*.

See Also

CGAL::ch_jarvis . . . . page 1367
CGAL::lower_hull_points_2 . . . . page 1393
CGAL::upper_hull_points_2 . . . . page 1395

Implementation

The function uses the Jarvis march (gift-wrapping) algorithm [Jar73]. This algorithm requires $O(nh)$ time in the worst case for n input points with h extreme points.

CGAL::ch_melkman

Definition

The function *ch_melkman* computes the counterclockwise sequence of extreme points of a sequence of points that forms a simple polyline or polygon.

#include <CGAL/ch_melkman.h>

template <class InputIterator, class OutputIterator>
OutputIterator ch_melkman(InputIterator first, InputIterator last, OutputIterator result, Traits ch_traits = Default_traits)

generates the counterclockwise sequence of extreme points of the points in the range [*first*, *beyond*). The resulting sequence is placed starting at position *result*, and the past-the-end iterator for the resulting sequence is returned.
Precondition: The source range [*first*,*beyond*) corresponds to a simple polyline. [*first*,*beyond*) does not contain *result*

.

The default traits class *Default_traits* is the kernel in which the type *InputIterator::value_type* is defined.

Requirements

1. *InputIterator::value_type* and *OutputIterator::value_type* are equivalent to *Traits::Point_2*.
2. *Traits* contains the following subset of types from the concept *ConvexHullTraits_2* and their corresponding member functions that return instances of these types:
   - *Traits::Point_2*,
   - *Traits::Equal_2*,
   - *Traits::Less_xy_2*,
   - *Traits::Left_turn_2*.

See Also

Implementation

It uses an implementation of Melkman's algorithm [Mel87]. Running time of this is linear.

CGAL::ch_nswe_point

Definition

The function *ch_nswe_point* finds the four extreme points of a given set of input points using a linear scan of the input points. That is, it determines the points with maximal y, minimal y, minimal x, and maximal x coordinates.

#include <CGAL/ch_selected_extreme_points_2.h>

template <class ForwardIterator>
void *ch_nswe_point(ForwardIterator first, ForwardIterator beyond, ForwardIterator& n, ForwardIterator& s, ForwardIterator& w, ForwardIterator& e, Traits ch_traits = Default_traits)*

traverses the range [*first*,*beyond*). After execution, the value of n is an iterator in the range such that $*n \geq_{yx} *it$ for all iterators it in the range. Similarly, for s, w, and e the inequalities $*s \leq_{yx} *it$, $*w \leq_{xy} *it$, and $*e \geq_{xy} *it$ hold for all iterators it in the range.

Requirements

Traits contains the following subset of types from the concept *ConvexHullTraits_2* and their corresponding member functions that return instances of these types:

- *Traits::Less_xy_2*,
- *Traits::Less_yx_2*.

The default traits class *Default_traits* is the kernel in which the type *ForwardIterator::value_type* is defined.

See Also

CGAL::ch_e_point . . . . page 1362
CGAL::ch_n_point . . . . page 1375
CGAL::ch_ns_point . . . . page 1374
CGAL::ch_s_point . . . . page 1376
CGAL::ch_w_point . . . . page 1378
CGAL::ch_we_point . . . . page 1377

Function

CGAL::ch_ns_point

Definition

The function *ch_ns_point* finds the points of a given set of input points with minimal and maximal x coordinates.

#include <CGAL/ch_selected_extreme_points_2.h>

template <class ForwardIterator>
void ch_ns_point(ForwardIterator first, ForwardIterator beyond, ForwardIterator& n, ForwardIterator& s, Traits ch_traits = Default_traits)

traverses the range [*first*,*beyond*). After execution, the value of n is an iterator in the range such that $*n \geq_{yx} *it$ for all iterators it in the range. Similarly, for s the inequality $*s \leq_{yx} *it$ holds for all iterators in the range.

The default traits class *Default_traits* is the kernel in which the type *ForwardIterator::value_type* is defined.

Requirements

Traits defines the type *Traits::Less_yx_2* as specified in the concept *ConvexHullTraits_2* and the corresponding member function that returns an instance of this type.

See Also

CGAL::ch_e_point . . . . page 1362
CGAL::ch_nswe_point . . . . page 1373
CGAL::ch_n_point . . . . page 1375
CGAL::ch_s_point . . . . page 1376
CGAL::ch_w_point . . . . page 1378
CGAL::ch_we_point . . . . page 1377

CGAL::ch_n_point

Definition

The function *ch_n_point* finds a point in a given set of input points with maximal y coordinate.

#include <CGAL/ch_selected_extreme_points_2.h>

template <class ForwardIterator>
void *ch_n_point(ForwardIterator first, ForwardIterator beyond, ForwardIterator& n, Traits ch_traits = Default_traits)*

traverses the range [*first*,*beyond*). After execution, the value of n is an iterator in the range such that $*n \geq_{yx} *it$ for all iterators *it* in the range.

The default traits class *Default_traits* is the kernel in which the type *ForwardIterator::value_type* is defined.

Requirements

Traits defines the type *Traits::Less_yx_2* as specified in the concept *ConvexHullTraits_2* and the corresponding member function that returns an instance of this type.

See Also

CGAL::ch_e_point . . . . page 1362
CGAL::ch_nswe_point . . . . page 1373
CGAL::ch_ns_point . . . . page 1374
CGAL::ch_s_point . . . . page 1376
CGAL::ch_w_point . . . . page 1378
CGAL::ch_we_point . . . . page 1377

Function

Function

CGAL::ch_s_point

Definition

The function *ch_s_point* finds a points in a given set of input points with minimal y coordinates.

#include <CGAL/ch_selected_extreme_points_2.h>

template <class ForwardIterator>
void *ch_s_point(ForwardIterator first,*
ForwardIterator beyond,
ForwardIterator& s,
Traits ch_traits = Default_traits)

traverses the range [*first*,*beyond*). After execution, the value of s is an iterator in the range such that $*s \leq_{yx} *it$ for all iterators *it* in the range.

The default traits class *Default_traits* is the kernel in which the type *ForwardIterator::value_type* is defined.

Requirements

Traits defines the type *Traits::Less_yx_2* as specified in the concept *ConvexHullTraits_2* and the corresponding member function that returns an instance of this type.

See Also

CGAL::ch_e_point . . . . . page 1362
CGAL::ch_nswe_point . . . . . page 1373
CGAL::ch_n_point . . . . . page 1375
CGAL::ch_ns_point . . . . . page 1374
CGAL::ch_w_point . . . . . page 1378
CGAL::ch_we_point . . . . . page 1377

CGAL::ch_we_point

Definition

The function *ch_we_point* finds two points of a given set of input points with minimal and maximal x coordinates.

#include <CGAL/ch_selected_extreme_points_2.h>

```
template <class ForwardIterator>
void    ch_we_point( ForwardIterator first,
                     ForwardIterator beyond,
                     ForwardIterator& w,
                     ForwardIterator& e,
                     Traits ch_traits = Default_traits)
```

traverses the range [*first*,*beyond*). After execution, the value of w is an iterator in the range such that $*w \leq_{xy} *it$ for all iterators *it* in the range. Similarly, for e the inequality $*e \geq_{xy} *it$ holds for all iterators in the range.

The default traits class *Default_traits* is the kernel in which the type *ForwardIterator::value_type* is defined.

Requirements

Traits defines the type *Traits::Less_xy_2* as specified in the concept *ConvexHullTraits_2* and the corresponding member function that returns an instance of this type.

See Also

CGAL::ch_e_point . . . . page 1362
CGAL::ch_nswe_point . . . . page 1373
CGAL::ch_n_point . . . . page 1375
CGAL::ch_ns_point . . . . page 1374
CGAL::ch_s_point . . . . page 1376
CGAL::ch_w_point . . . . page 1378

Function

CGAL::ch_w_point

Definition

The function *ch_w_point* finds a point in a given set of input points with minimal x coordinate.

#include <CGAL/ch_selected_extreme_points_2.h>

template <class ForwardIterator>
void *ch_w_point(ForwardIterator first, ForwardIterator beyond, ForwardIterator& w, Traits ch_traits = Default_traits)*

traverses the range [*first*,*beyond*). After execution, the value of w is an iterator in the range such that $*w \leq_{xy} *it$ for all iterators *it* in the range.

Requirements

Traits defines the type *Traits::Less_xy_2* as specified in the concept *ConvexHullTraits_2* and the corresponding member function that returns an instance of this type.

The default traits class *Default_traits* is the kernel in which the type *ForwardIterator::value_type* is defined.

See Also

CGAL::ch_e_point . . . . page 1362
CGAL::ch_nswe_point . . . . page 1373
CGAL::ch_n_point . . . . page 1375
CGAL::ch_ns_point . . . . page 1374
CGAL::ch_s_point . . . . page 1376
CGAL::ch_we_point . . . . page 1377

CGAL::convex_hull_2

Function

Definition

The function *convex_hull_2* generates the counterclockwise sequence of extreme points from a given set of input points.

#include <CGAL/convex_hull_2.h>

template <class InputIterator, class OutputIterator>
OutputIterator convex_hull_2(InputIterator first,
InputIterator beyond,
OutputIterator result,
Traits ch_traits = Default_traits)

generates the counterclockwise sequence of extreme points of the points in the range [*first*,*beyond*). The resulting sequence is placed starting at position *result*, and the past-the-end iterator for the resulting sequence is returned. It is not specified at which point the cyclic sequence of extreme points is cut into a linear sequence.
Precondition: The source range [*first*,*beyond*) does not contain *result*.

The default traits class *Default_traits* is the kernel in which the type *InputIterator::value_type* is defined.

Requirements

1. *InputIterator::value_type* and *OutputIterator::value_type* are equivalent to *Traits::Point_2*.
2. *Traits* contains the following subset of types from the concept *ConvexHullTraits_2* and their corresponding member functions that return instances of these types:
   - *Traits::Point_2*,
   - *Traits::Less_signed_distance_to_line_2*,
   - *Traits::Equal_2*,
   - *Traits::Less_xy_2*,
   - *Traits::Less_yx_2*,
   - *Traits::Left_turn_2*.

See Also

CGAL::ch_akl_toussaint . . . . page 1356
CGAL::ch_bykat . . . . page 1358
CGAL::ch_eddy . . . . page 1360
CGAL::ch_graham_andrew . . . . page 1363
CGAL::ch_jarvis . . . . page 1367
CGAL::ch_melkman . . . . page 1371

Implementation

One of two algorithms is used, depending on the type of iterator used to specify the input points. For input iterators, the algorithm used is that of Bykat [Byk78], which has a worst-case running time of $O(nh)$, where n is the number of input points and h is the number of extreme points. For all other types of iterators, the $O(n \log n)$ algorithm of of Akl and Toussaint [AT78] is used.

ConvexHullTraits_2

Definition

All convex hull and extreme point algorithms provided in CGAL are parameterized with a traits class *Traits*, which defines the primitives (objects and predicates) that the convex hull algorithms use. ConvexHullTraits_2 defines the complete set of primitives required in these functions. The specific subset of these primitives required by each function is specified with each function.

Types

ConvexHullTraits_2:: Point_2 — The point type on which the convex hull functions operate.

ConvexHullTraits_2:: Equal_2 — Binary predicate object type comparing *Point_2*s. Must provide *bool operator()(Point_2 p, Point_2 q)* where *true* is returned iff $p ==_{xy} q$, false otherwise.

ConvexHullTraits_2:: Less_xy_2 — Binary predicate object type comparing *Point_2*s lexicographically. Must provide *bool operator()(Point_2 p, Point_2 q)* where *true* is returned iff $p <_{xy} q$. We have $p <_{xy} q$, iff $p_x < q_x$ or $p_x = q_x$ and $p_y < q_y$, where p_x and p_y denote x and y coordinate of point p, respectively.

ConvexHullTraits_2:: Less_yx_2 — Same as *Less_xy_2* with the roles of x and y interchanged.

ConvexHullTraits_2:: Left_turn_2 — Predicate object type that must provide *bool operator()(Point_2 p,Point_2 q,Point_2 r)*, which returns *true* iff r lies to the left of the oriented line through p and q.

ConvexHullTraits_2:: Less_signed_distance_to_line_2

Predicate object type that must provide *bool operator()(Point_2 p, Point_2 q, Point_2 r,Point_2 s)*, which returns *true* iff the signed distance from r to the line l_{pq} through p and q is smaller than the distance from s to l_{pq}. It is used to compute the point right of a line with maximum unsigned distance to the line. The predicate must provide a total order compatible with convexity, *i.e.*, for any line segment s one of the endpoints of s is the smallest point among the points on s, with respect to the order given by *Less_signed_distance_to_line_2*.

ConvexHullTraits_2:: Less_rotate_ccw_2

Predicate object type that must provide *bool operator()(Point_2 e, Point_2 p,Point_2 q)*, where *true* is returned iff a tangent at e to the point set $\{e,p,q\}$ hits p before q when rotated counterclockwise around e. Ties are broken such that the point with larger distance to e is smaller!

Creation

Only a copy constructor is required.

ConvexHullTraits_2 traits(& t);

Operations

The following member functions to create instances of the above predicate object types must exist.

Equal_2	*traits.equal_2_object()*
Less_xy_2	*traits.less_xy_2_object()*
Less_yx_2	*traits.less_yx_2_object()*
Less_signed_distance_to_line_2	
	traits.less_signed_distance_to_line_2_object()
Less_rotate_ccw_2	*traits.less_rotate_ccw_2_object()*
Left_turn_2	*traits.left_turn_2_object()*

Has Models

CGAL::Convex_hull_constructive_traits_2<R> . . . . page 1383
CGAL::Convex_hull_traits_2<R> . . . . page 1390
CGAL::Projection_traits_xy_3<K> . . . . page 604
CGAL::Projection_traits_yz_3<K>
CGAL::Projection_traits_zx_3<K>

deprecated

CGAL::Convex_hull_projective_xy_traits_2<Point_3> . . . . page 1384
CGAL::Convex_hull_projective_xz_traits_2<Point_3> . . . . page 1386
CGAL::Convex_hull_projective_yz_traits_2<Point_3> . . . . page 1388

deprecated

See Also

IsStronglyConvexTraits_3 . . . . page 1424

CGAL::Convex_hull_constructive_traits_2<R>

Class

Definition

The class *Convex_hull_constructive_traits_2<R>* serves as a traits class for all the two-dimensional convex hull and extreme point calculation function. Unlike the class *CGAL::Convex_hull_traits_2<R>*, this class makes use of previously computed results to avoid redundancy. For example, in the sidedness tests, lines (of type *R::Line_2*) are constructed, which is equivalent to the precomputation of subdeterminants of the orientation-determinant for three points.

#include <CGAL/convex_hull_constructive_traits_2.h>

Is Model for the Concepts

ConvexHullTraits_2 . . . . page 1381

Types

typedef R::Point_2	*Point_2;*
typedef R::Less_xy_2	*Less_xy_2;*
typedef R::Less_yx_2	*Less_yx_2;*
typedef CGAL::r_Less_dist_to_line<R>	*Less_signed_distance_to_line_2;*
typedef R::Less_rotate_ccw	*Less_rotate_ccw_2;*
typedef R::Left_turn_2	*Left_turn_2;*
typedef R::Equal_2	*Equal_2;*

Creation

Convex_hull_constructive_traits_2<R> traits;	default constructor.

Operations

Less_xy_2	*traits.less_xy_2_object()*
Less_yx_2	*traits.less_yx_2_object()*
Less_signed_distance_to_line_2	*traits.less_signed_distance_to_line_2_object()*
Less_rotate_ccw_2	*traits.less_rotate_ccw_2_object()*
Left_turn_2	*traits.left_turn_2_object()*
Equal_2	*traits.equal_2_object()*

See Also

CGAL::Convex_hull_projective_xy_traits_2<Point_3> . . . . page 1384
CGAL::Convex_hull_projective_xz_traits_2<Point_3> . . . . page 1386
CGAL::Convex_hull_projective_yz_traits_2<Point_3> . . . . page 1388
CGAL::Convex_hull_traits_2<R> . . . . page 1390

Class

CGAL::Convex_hull_projective_xy_traits_2<Point_3>

The functionality of this class has been generalized to other packages than 2D convex hulls. The more general class *Projection_traits_xy_3* can be found in the 2D and 3D Linear Geometric Kernel. Note that the deprecated class was templated by a point class, whereas the new class is templated by a geometric kernel.

deprecated

Definition

The class *Convex_hull_projective_xy_traits_2<Point_3>* serves as a traits class for all the two-dimensional convex hull and extreme point calculation function. This class can be used to compute the convex hull of a set of 3D points projected onto the *xy* plane (*i.e.*, by ignoring the *z* coordinate).

#include <CGAL/Convex_hull_projective_xy_traits_2.h>

Is Model for the Concepts

ConvexHullTraits_2 . . . . page 1381

Types

typedef Point_3	*Point_2;*
typedef Less_xy_plane_xy_2<Point_3>	*Less_xy_2;*
typedef Less_yx_plane_xy_2<Point_3>	*Less_yx_2;*
typedef Less_dist_to_line_plane_xy_2<Point_3>	*Less_signed_distance_to_line_2;*
typedef Less_rotate_ccw_plane_xy_2<Point_3>	*Less_rotate_ccw_2;*
typedef Left_turn_plane_xy_2<Point_3>	*Left_turn_2;*
typedef Equal_xy_plane_xy_2<Point_3>	*Equal_2;*

Creation

Convex_hull_projective_xy_traits_2<Point_3> traits; default constructor.

Operations

Less_xy_2	*traits.less_xy_2_object()*
Less_yx_2	*traits.less_yx_2_object()*
Less_signed_distance_to_line_2	*traits.less_signed_distance_to_line_2_object()*
Less_rotate_ccw_2	*traits.less_rotate_ccw_2_object()*
Left_turn_2	*traits.left_turn_2_object()*
Equal_2	*traits.equal_2_object()*

See Also

CGAL::Convex_hull_constructive_traits_2<R> . . . . page 1383
CGAL::Convex_hull_projective_xz_traits_2<Point_3> . . . . page 1386

deprecated

Class

CGAL::Convex_hull_projective_xz_traits_2<Point_3>

The functionality of this class has been generalized to other packages than 2D convex hulls. The more general class *Projection_traits_xy_3* can be found in the 2D and 3D Linear Geometric Kernel. Note that the deprecated class was templated by a point class, whereas the new class is templated by a geometric kernel.

deprecated

Definition

The class *Convex_hull_projective_xz_traits_2<Point_3>* serves as a traits class for all the two-dimensional convex hull and extreme point calculation function. This class can be used to compute the convex hull of a set of 3D points projected onto the *xz* plane (*i.e.*, by ignoring the *y* coordinate).

#include <CGAL/Convex_hull_projective_xz_traits_2.h>

Is Model for the Concepts

ConvexHullTraits_2 . . . . page 1381

Types

typedef Point_3	*Point_2;*
typedef Less_xy_plane_xz_2<Point_3>	*Less_xy_2;*
typedef Less_yx_plane_xz_2<Point_3>	*Less_yx_2;*
typedef Less_dist_to_line_plane_xz_2<Point_3>	*Less_signed_distance_to_line_2;*
typedef Less_rotate_ccw_plane_xz_2<Point_3>	*Less_rotate_ccw_2;*
typedef Left_turn_plane_xz_2<Point_3>	*Left_turn_2;*
typedef Equal_xy_plane_xz_2<Point_3>	*Equal_2;*

Creation

Convex_hull_projective_xz_traits_2<Point_3> traits; default constructor.

Operations

Less_xy_2	*traits.less_xy_2_object()*
Less_yx_2	*traits.less_yx_2_object()*
Less_signed_distance_to_line_2	*traits.less_signed_distance_to_line_2_object()*
Less_rotate_ccw_2	*traits.less_rotate_ccw_2_object()*
Left_turn_2	*traits.left_turn_2_object()*
Equal_2	*traits.equal_2_object()*

See Also

CGAL::Convex_hull_constructive_traits_2<R> . . . . page 1383
CGAL::Convex_hull_projective_xy_traits_2<Point_3> . . . . page 1384

deprecated

Class

CGAL::Convex_hull_projective_yz_traits_2<Point_3>

The functionality of this class has been generalized to other packages than 2D convex hulls. The more general class *Projection_traits_xy_3* can be found in the 2D and 3D Linear Geometric Kernel. Note that the deprecated class was templated by a point class, whereas the new class is templated by a geometric kernel.

deprecated

Definition

The class *Convex_hull_projective_yz_traits_2<Point_3>* serves as a traits class for all the two-dimensional convex hull and extreme point calculation function. This class can be used to compute the convex hull of a set of 3D points projected onto the *yz* plane (*i.e.*, by ignoring the *x* coordinate).

#include <CGAL/Convex_hull_projective_yz_traits_2.h>

Is Model for the Concepts

ConvexHullTraits_2 . . . . page 1381

Types

typedef Point_3	*Point_2;*
typedef Less_xy_plane_yz_2<Point_3>	*Less_xy_2;*
typedef Less_yx_plane_yz_2<Point_3>	*Less_yx_2;*
typedef Less_dist_to_line_plane_yz_2<Point_3>	*Less_signed_distance_to_line_2;*
typedef Less_rotate_ccw_plane_yz_2<Point_3>	*Less_rotate_ccw_2;*
typedef Left_turn_plane_yz_2<Point_3>	*Left_turn_2;*
typedef Equal_xy_plane_yz_2<Point_3>	*Equal_2;*

Creation

Convex_hull_projective_yz_traits_2<Point_3> traits; default constructor.

Operations

Less_xy_2	*traits.less_xy_2_object()*
Less_yx_2	*traits.less_yx_2_object()*
Less_signed_distance_to_line_2	*traits.less_signed_distance_to_line_2_object()*
Less_rotate_ccw_2	*traits.less_rotate_ccw_2_object()*
Left_turn_2	*traits.left_turn_2_object()*
Equal_2	*traits.equal_2_object()*

See Also

CGAL::Convex_hull_constructive_traits_2<R> . . . . page 1383
CGAL::Convex_hull_projective_xz_traits_2<Point_3> . . . . page 1386

deprecated

Class

CGAL::Convex_hull_traits_2<R>

Definition

The class *Convex_hull_traits_2<R>* serves as a traits class for all the two-dimensional convex hull and extreme point calculation function. This class corresponds to the default traits class for these functions.

#include <CGAL/convex_hull_traits_2.h>

Is Model for the Concepts

ConvexHullTraits_2 . . . . page 1381

Types

typedef R::Point_2	*Point_2;*
typedef R::Less_xy	*Less_xy_2;*
typedef R::Less_yx	*Less_yx_2;*
typedef R::Less_signed_distance_to_line_2	*Less_signed_distance_to_line_2;*
typedef R::Less_rotate_ccw_2	*Less_rotate_ccw_2;*
typedef R::Left_turn_2	*Left_turn_2;*
typedef R::Equal_2	*Equal_2;*

Creation

Convex_hull_traits_2<R> traits(Convex_hull_traits_2& t); copy constructor.

Operations

Less_xy_2	*traits.less_xy_2_object()*
Less_yx_2	*traits.less_yx_2_object()*
Less_signed_distance_to_line_2	*traits.less_signed_distance_to_line_2_object()*
Less_rotate_ccw_2	*traits.less_rotate_ccw_2_object()*
Left_turn_2	*traits.left_turn_2_object()*
Equal_2	*traits.equal_2_object()*

See Also

CGAL::Convex_hull_constructive_traits_2<R> . . . . page 1383
CGAL::Convex_hull_projective_xy_traits_2<Point_3> . . . . page 1384
CGAL::Convex_hull_projective_xz_traits_2<Point_3> . . . . page 1386
CGAL::Convex_hull_projective_yz_traits_2<Point_3> . . . . page 1388

CGAL::is_ccw_strongly_convex_2

Function

Definition

The function *is_ccw_strongly_convex_2* determines if a given sequence of points defines a counterclockwise-oriented, strongly convex polygon. A set of points is said to be strongly convex if it consists of only extreme points (*i.e.*, vertices of the convex hull).

#include <CGAL/convexity_check_2.h>

template <class ForwardIterator, class Traits>
bool *is_ccw_strongly_convex_2(ForwardIterator first, ForwardIterator beyond, Traits ch_traits = Default_traits)*

returns *true*, iff the point elements in [*first*,*beyond*) form a counterclockwise-oriented strongly convex polygon.

The default traits class *Default_traits* is the kernel in which the type *ForwardIterator::value_type* is defined.

Requirements

Traits contains the following subset of types from the concept *ConvexHullTraits_2* and their corresponding member functions that return instances of these types:

- *Traits::Less_xy_2*,
- *Traits::Equal_2*,
- *Traits::Left_turn_2*.

See Also

CGAL::is_cw_strongly_convex_2 . . . . page 1392
CGAL::is_strongly_convex_3 . . . . page 1423

Implementation

The algorithm requires $O(n)$ time for a set of n input points.

Function

CGAL::is_cw_strongly_convex_2

Definition

The function *is_cw_strongly_convex_2* determines if a given sequence of points defines a clockwise-oriented, strongly convex polygon. A set of points is said to be strongly convex if it consists of only extreme points (*i.e.*, vertices of the convex hull).

#include <CGAL/convexity_check_2.h>

template <class ForwardIterator, class Traits>
bool *is_cw_strongly_convex_2(ForwardIterator first, ForwardIterator beyond, Traits ch_traits = Default_traits)*

returns *true*, iff the point elements in [*first*,*beyond*) form a clockwise-oriented strongly convex polygon.

The default traits class *Default_traits* is the kernel in which the type *ForwardIterator::value_type* is defined.

Requirements

Traits contains the following subset of types from the concept *ConvexHullTraits_2* and their corresponding member functions that return instances of these types:

- *Traits::Equal_2*,
- *Traits::Less_xy_2*,
- *Traits::Left_turn_2*.

See Also

CGAL::is_ccw_strongly_convex_2 . . . . page 1391
CGAL::is_strongly_convex_3 . . . . page 1423

Implementation

The algorithm requires $O(n)$ time for a set of n input points.

CGAL::lower_hull_points_2

Definition

The function *lower_hull_points_2* generates the counterclockwise sequence of extreme points on the lower hull of a given set of input points.

#include <CGAL/convex_hull_2.h>

template <class InputIterator, class OutputIterator>
OutputIterator lower_hull_points_2(InputIterator first, InputIterator beyond, OutputIterator result, Traits ch_traits = Default_traits)

generates the counterclockwise sequence of extreme points on the lower hull of the points in the range [*first*, *beyond*). The resulting sequence is placed starting at position *result*, and the past-the-end iterator for the resulting sequence is returned. The sequence starts with the leftmost point; the rightmost point is not included. If there is only one extreme point (*i.e.*, leftmost and rightmost point are equal) the extreme point is reported.
Precondition: The source range [*first*,*beyond*) does not contain *result*.

The default traits class *Default_traits* is the kernel in which the type *InputIterator::value_type* is defined.

The different treatment by *CGAL::upper_hull_points_2* of the case that all points are equal ensures that concatenation of lower and upper hull points gives the sequence of extreme points.

Requirements

1. *InputIterator::value_type* and *OutputIterator::value_type* are equivalent to *Traits::Point_2*.
2. *Traits* contains the following subset of types from the concept *ConvexHullTraits_2* and their corresponding member functions that return instances of these types:
   - *Traits::Point_2*,
   - *Traits::Equal_2*,
   - *Traits::Less_xy_2*,
   - *Traits::Less_yx_2*,
   - *Traits::Left_turn_2*.

See Also

CGAL::ch_graham_andrew . . . . page 1363
CGAL::ch_graham_andrew_scan . . . . page 1365
CGAL::upper_hull_points_2 . . . . page 1395

Implementation

This function uses Andrew's variant of Graham's scan algorithm [And79, Meh84]. The algorithm has worst-case running time of $O(n \log n)$ for n input points.

CGAL::upper_hull_points_2

Function

Definition

The function *upper_hull_points_2* generates the counterclockwise sequence of extreme points on the upper hull of a given set of input points.

#include <CGAL/convex_hull_2.h>

```
template <class InputIterator, class OutputIterator>
OutputIterator     upper_hull_points_2( InputIterator first,
                                        InputIterator beyond,
                                        OutputIterator result,
                                        Traits ch_traits = Default_traits)
```

generates the counterclockwise sequence of extreme points on the upper hull of the points in the range [*first*, *beyond*). The resulting sequence is placed starting at position *result*, and the past-the-end iterator for the resulting sequence is returned. The sequence starts with the rightmost point, the leftmost point is not included. If there is only one extreme point (*i.e.*, the leftmost and rightmost point are equal), the extreme point is not reported.
Precondition: The source range [*first*,*beyond*) does not contain *result*.

The default traits class *Default_traits* is the kernel in which the type *InputIterator::value_type* is defined.

The different treatment by *CGAL::lower_hull_points_2* of the case that all points are equal ensures that concatenation of lower and upper hull points gives the sequence of extreme points.

Requirements

1. *InputIterator::value_type* and *OutputIterator::value_type* are equivalent to *Traits::Point_2*.
2. *Traits* contains the following subset of types from the concept *ConvexHullTraits_2* and their corresponding member functions that return instances of these types:
   - *Traits::Point_2*,
   - *Traits::Equal_2*,
   - *Traits::Less_xy_2*,
   - *Traits::Less_yx_2*,
   - *Traits::Left_turn_2*.

See Also

CGAL::ch_graham_andrew . . . . page 1363
CGAL::ch_graham_andrew_scan . . . . page 1365
CGAL::lower_hull_points_2 . . . . page 1393

Implementation

This function uses Andrew's variant of Graham's scan algorithm [And79, Meh84]. The algorithm has worst-case running time of $O(n \log n)$ for n input points.

Chapter 16

3D Convex Hulls

Susan Hert and Stefan Schirra

Contents

16.1 Introduction

A subset $S \subseteq \mathbb{R}^3$ is convex if for any two points p and q in the set the line segment with endpoints p and q is contained in S. The convex hull of a set S is the smallest convex set containing S. The convex hull of a set of points $P \in \mathbb{R}^3$ is a convex polytope with vertices in P. A point in P is an extreme point (with respect to P) if it is a vertex of the convex hull of P. A set of points is said to be strongly convex if it consists of only extreme points.

This chapter describes the functions provided in CGAL for producing convex hulls in three dimensions as well as functions for checking if sets of points are strongly convex are not. One can compute the convex hull of a set of points in three dimensions in one of three ways in CGAL: using a static algorithm, using an incremental construction algorithm, or using a triangulation to get a fully dynamic computation.

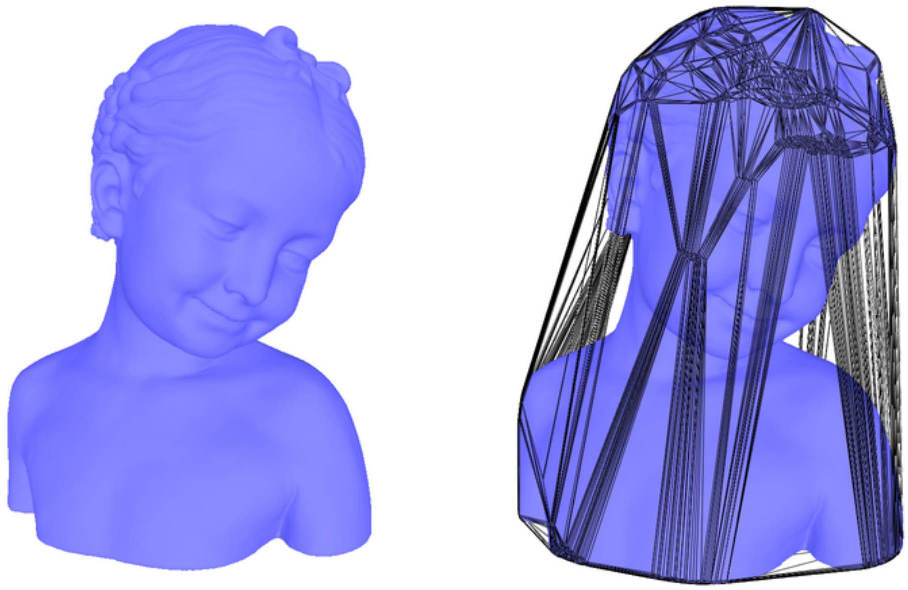


Figure 16.1: The convex hull of a model made of 192135 points.

16.2 Static Convex Hull Construction

The function *convex_hull_3* provides an implementation of the quickhull algorithm [BDH96] for three dimensions. There are two versions of this function available, one that can be used when it is known that the output will be a polyhedron (*i.e.*, there are more than three points and they are not all collinear) and one that handles all degenerate cases and returns a *CGAL::Object*, which may be a point, a segment, a triangle, or a polyhedron. Both versions accept a range of input iterators defining the set of points whose convex hull is to be computed and a traits class defining the geometric types and predicates used in computing the hull.

16.2.1 Traits Class

The function *convex_hull_3* is parameterized by a traits class, which specifies the types and geometric primitives to be used in the computation. If input points from a kernel with exact predicates and non-exact constructions are used, and a certified result is expected, the traits *Convex_hull_traits_3<R>* should be used (*R* being the input kernel). Note that the default traits class takes this into account.

16.2.2 Convexity Checking

The function *is_strongly_convex_3* implements the algorithm of Mehlhorn *et al.* [MNS+96] to determine if the vertices of a given polytope constitute a strongly convex point set or not. This function is used in postcondition testing for *convex_hull_3*.

16.2.3 Example

The following program computes the convex hull of a set of 250 random points chosen from a sphere of radius 100. We assume that the points are not all identical and not all collinear, thus we directly use a polyhedron as output. Note the usage of the functor *Plane_from_facet* together with *std::transform* to compute the equations of the plane of each facet of the convex hull.

```
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/point_generators_3.h>
#include <CGAL/algorithm.h>
#include <CGAL/Polyhedron_3.h>
#include <CGAL/convex_hull_3.h>
#include <vector>


typedef CGAL::Exact_predicates_inexact_constructions_kernel  K;
typedef CGAL::Polyhedron_3<K>                                Polyhedron_3;
typedef K::Segment_3                                         Segment_3;

// define point creator
typedef K::Point_3                                           Point_3;
typedef CGAL::Creator_uniform_3<double, Point_3>  PointCreator;

//a functor computing the plane containing a triangular facet
struct Plane_from_facet {
  Polyhedron_3::Plane_3 operator()(Polyhedron_3::Facet& f) {
      Polyhedron_3::Halfedge_handle h = f.halfedge();
      return Polyhedron_3::Plane_3( h->vertex()->point(),
                                    h->next()->vertex()->point(),
                                    h->opposite()->vertex()->point());
  }
};


int main()
{
  CGAL::Random_points_in_sphere_3<Point_3, PointCreator> gen(100.0);

  // generate 250 points randomly on a sphere of radius 100.0
  // and copy them to a vector
  std::vector<Point_3> points;
  CGAL::cpp0x::copy_n( gen, 250, std::back_inserter(points) );

  // define polyhedron to hold convex hull
  Polyhedron_3 poly;

  // compute convex hull of non-collinear points
  CGAL::convex_hull_3(points.begin(), points.end(), poly);

  std::cout << "The convex hull contains " << poly.size_of_vertices() <<
   " vertices" << std::endl;

  // assign a plane equation to each polyhedron facet using functor
   Plane_from_facet
```

```
  std::transform( poly.facets_begin(), poly.facets_end(),
   poly.planes_begin(),Plane_from_facet());


  return 0;
}
```

File: examples/Convex_hull_3/quickhull_3.cpp

16.3 Incremental Convex Hull Construction

The function *convex_hull_incremental_3* provides an interface similar to *convex_hull_3* for the d-dimensional incremental construction algorithm [CMS93] implemented by the class *CGAL::Convex_hull_d<R>* that is specialized to three dimensions. This function accepts an iterator range over a set of input points and returns a polyhedron, but it does not have a traits class in its interface. It uses the kernel class *Kernel* used in the polyhedron type to define an instance of the adapter traits class *CGAL::Convex_hull_d_traits_3<Kernel>*.

In almost all cases, the static and the dynamic version will be faster than the incremental convex hull algorithm (mainly because of the lack of efficient filtering and the overhead of the general d-dimension). The incremental version is provided for completeness and educational purposes. You should use the dynamic version when you need an efficient incremental convex hull algorithm.

To use the full functionality available with the d-dimensional class *CGAL::Convex_hull_d<R>* in three dimensions (*e.g.*, the ability to insert new points and to query if a point lies in the convex hull or not), you can instantiate the class *CGAL::Convex_hull_d<K>* with the adapter traits class *CGAL::Convex_hull_d_traits_3<K>*, as shown in the following example.

16.3.1 Example

```
#include <CGAL/Cartesian.h>
#include <CGAL/point_generators_3.h>
#include <CGAL/Convex_hull_d.h>
#include <CGAL/Convex_hull_d_traits_3.h>
#include <CGAL/Convex_hull_d_to_polyhedron_3.h>
#include <CGAL/Polyhedron_3.h>
#include <CGAL/algorithm.h>
#include <vector>
#include <cassert>

#ifdef CGAL_USE_GMP
#include <CGAL/Gmpq.h>
typedef CGAL::Gmpq RT;
#else
#include <CGAL/MP_Float.h>
typedef CGAL::Quotient<CGAL::MP_Float> RT;
#endif
```

```
typedef CGAL::Cartesian<RT>                             K;
typedef K::Point_3                                      Point_3;
typedef CGAL::Polyhedron_3< K>                          Polyhedron_3;

typedef CGAL::Convex_hull_d_traits_3<K>                 Hull_traits_3;
typedef CGAL::Convex_hull_d< Hull_traits_3 >            Convex_hull_3;
typedef CGAL::Creator_uniform_3<double, Point_3>        Creator;

int main ()
{
  Convex_hull_3 CH(3);   // create instance of the class with dimension ==
     3

  // generate 250 points randomly on a sphere of radius 100
  // and insert them into the convex hull
  CGAL::Random_points_in_sphere_3<Point_3, Creator> gen(100);

  for (int i = 0; i < 250 ; i++, ++gen)
     CH.insert(*gen);

  assert(CH.is_valid());

  // define polyhedron to hold convex hull and create it
  Polyhedron_3 P;
  CGAL::convex_hull_d_to_polyhedron_3(CH,P);

  std::cout << "The convex hull has " << P.size_of_vertices()
            << " vertices" << std::endl;
  return 0;
}
```

File: examples/Convex_hull_3/incremental_hull_class_3.cpp

16.4 Dynamic Convex Hull Construction

Fully dynamic maintenance of a convex hull can be achieved by using the class *CGAL::Delaunay_triangulation_3*. This class supports insertion and removal of points (*i.e.*, vertices of the triangulation) and the convex hull edges are simply the finite edges of infinite faces. The following example illustrates the dynamic construction of a convex hull. First, random points from a sphere of a certain radius are generated and are inserted into a triangulation. Then the number of points of the convex hull are obtained by counting the number of triangulation vertices incident to the infinite vertex. Some of the points are removed and then the number of points remaining on the hull are determined. Notice that the vertices incident to the infinite vertex of the triangulation are on the convex hull but it may be that not all of them are vertices of the hull.

16.4.1 Example

```
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/point_generators_3.h>
#include <CGAL/Delaunay_triangulation_3.h>
```

```
#include <CGAL/Polyhedron_3.h>
#include <CGAL/convex_hull_3_to_polyhedron_3.h>
#include <CGAL/algorithm.h>

#include <list>

typedef CGAL::Exact_predicates_inexact_constructions_kernel      K;
typedef K::Point_3                                              Point_3;
typedef CGAL::Delaunay_triangulation_3<K>                       Delaunay;
typedef Delaunay::Vertex_handle
    Vertex_handle;
typedef CGAL::Polyhedron_3<K>
    Polyhedron_3;

int main()
{
  CGAL::Random_points_in_sphere_3<Point_3> gen(100.0);
  std::list<Point_3>   points;

  // generate 250 points randomly on a sphere of radius 100.0
  // and insert them into the triangulation
  CGAL::cpp0x::copy_n(gen, 250, std::back_inserter(points) );
  Delaunay T;
  T.insert(points.begin(), points.end());

  std::list<Vertex_handle>  vertices;
  T.incident_vertices(T.infinite_vertex(), std::back_inserter(vertices));
  std::cout << "This convex hull of the 250 points has "
            << vertices.size() << " points on it." << std::endl;

  // remove 25 of the input points
  std::list<Vertex_handle>::iterator v_set_it = vertices.begin();
  for (int i = 0; i < 25; i++)
  {
     T.remove(*v_set_it);
     v_set_it++;
  }

  //copy the convex hull of points into a polyhedron and use it
  //to get the number of points on the convex hull
  Polyhedron_3 chull;
  CGAL::convex_hull_3_to_polyhedron_3(T,chull);

  std::cout << "After removal of 25 points, there are "
            << chull.size_of_vertices() << " points on the convex hull."
   << std::endl;



  return 0;
}
```

File: examples/Convex_hull_3/dynamic_hull_3.cpp

16.5 Performance

In the following, we compare the running times of the three approaches to compute 3D convex hulls. For the static version (using *CGAL::convex_hull_3*) and the dynamic version (using *CGAL::Delaunay_triangulation_3* and *CGAL::convex_hull_3_to_polyhedron_3*), the kernel used was *CGAL::Exact_predicates_inexact_constructions_kernel*. For the incremental version (using *CGAL::convex_hull_incremental_3*), the kernel used was *CGAL::Exact_predicates_exact_constructions_kernel*.

To compute the convex hull of a million of random points in a unit ball the static approach needed 1.63s, while the dynamic and incremental approaches needed 9.50s and 11.54s respectively. To compute the convex hull of the model of Figure 16.1 featuring 192135 points, the static approach needed 0.18s, while the dynamic and incremental approaches needed 1.90s and 6.80s respectively.

The measurements have been performed using CGAL 3.9, using the GNU C++ compiler version 4.3.5, under Linux (Debian distribution), with the compilation options `-O3 -DCGAL_NDEBUG`. The computer used was equipped with a 64bit Intel Xeon 2.27GHz processor and 12GB of RAM.

3D Convex Hulls
Reference Manual

Susan Hert and Stefan Schirra

A subset $S \subseteq \mathbb{R}^3$ is convex if for any two points p and q in the set the line segment with endpoints p and q is contained in S. The convex hull of a set S is the smallest convex set containing S. The convex hull of a set of points P is a convex polytope with vertices in P. A point in P is an extreme point (with respect to P) if it is a vertex of the convex hull of P.

CGAL provides functions for computing convex hulls in two, three and arbitrary dimensions as well as functions for testing if a given set of points in is strongly convex or not. This chapter describes the functions available for three dimensions.

Assertions

The assertion flags for the convex hull and extreme point algorithms use *CH* in their names (*e.g.*, *CGAL_CH_NO_POSTCONDITIONS*). For the convex hull algorithms, the postcondition check tests only convexity (if not disabled), but not containment of the input points in the polygon or polyhedron defined by the output points. The latter is considered an expensive checking and can be enabled by defining *CGAL_CH_CHECK_EXPENSIVE*.

16.6 Classified Reference Pages

Concepts

Traits Classes

Convex Hull Functions

Convexity Checking Function

16.7 Alphabetical List of Reference Pages

CGAL::convex_hull_3

Definition

The function *convex_hull_3* computes the convex hull of a given set of three-dimensional points Two versions of this function are available. The first can be used when it is known that the result will be a polyhedron and the second when a degenerate hull may also be possible.

#include <CGAL/convex_hull_3.h>

template <class InputIterator, class Polyhedron_3, class Traits>
void convex_hull_3(InputIterator first, InputIterator last, Polyhedron_3& P, Traits ch_traits = Default_traits)

computes the convex hull of the set of points in the range [*first*, *last*). The polyhedron *P* is cleared, then the convex hull is stored in *P* and the plane equations of each face are not computed.
Precondition: : There are at least four points in the range [*first*, *last*) not all of which are collinear.

template <class InputIterator, class Traits>
void convex_hull_3(InputIterator first, InputIterator last, Object& ch_object, Traits ch_traits = Default_traits)

computes the convex hull of the set of points in the range [*first*, *last*). The result, which may be a point, a segment, a triangle, or a polyhedron, is stored in *ch_object*. When the result is a polyhedron, the plane equations of each face are not computed.

Requirements

Both functions require the following:

1. *InputIterator::value_type* is equivalent to *Traits::Point_3*.
2. *Traits* is a model of the concept *ConvexHullTraits_3* . For the purposes of checking the postcondition that the convex hull is valid, *Traits* should also be a model of the concept *IsStronglyConvexTraits_3*.

Both functions have an additional requirement for the polyhedron that is to be constructed. For the first version this is that:

- *Polyhedron_3* is a model of *ConvexHullPolyhedron_3*, and for the second, it is required that

- *Traits* defines a type *Polyhedron_3* that is a model of *ConvexHullPolyhedron_3*.

For both versions, if the kernel *R* of the points determined by *InputIterator::value_type* is a kernel with exact predicates but inexact constructions (in practice we check *R::Has_filtered_predicates_tag* is *Tag_true* and *R::FT* is a floating point type), then the default traits class of *convex_hull_3* is *Convex_hull_traits_3*<*R*>, and *R* otherwise.

See Also

CGAL::convex_hull_incremental_3 . . . . page 1411
CGAL::ch_eddy . . . . page 1360
CGAL::convex_hull_2 . . . . page 1379

Implementation

The algorithm implemented by these functions is the quickhull algorithm of Barnard *et al.* [BDH96].

Example

The following program computes the convex hull of a set of 250 random points chosen from a sphere of radius 100. It then determines if the resulting hull is a segment or a polyhedron. Notice that the traits class is not necessary in the call to *convex_hull_3* but is used in the definition of *Polyhedron_3*.

```
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/point_generators_3.h>
#include <CGAL/algorithm.h>
#include <CGAL/Polyhedron_3.h>
#include <CGAL/convex_hull_3.h>
#include <vector>


typedef CGAL::Exact_predicates_inexact_constructions_kernel  K;
typedef CGAL::Polyhedron_3<K>                               Polyhedron_3;
typedef K::Segment_3                                        Segment_3;

// define point creator
typedef K::Point_3                                          Point_3;
typedef CGAL::Creator_uniform_3<double, Point_3>  PointCreator;

//a functor computing the plane containing a triangular facet
struct Plane_from_facet {
  Polyhedron_3::Plane_3 operator()(Polyhedron_3::Facet& f) {
      Polyhedron_3::Halfedge_handle h = f.halfedge();
      return Polyhedron_3::Plane_3( h->vertex()->point(),
                                    h->next()->vertex()->point(),
                                    h->opposite()->vertex()->point());
  }
};
```

```
int main()
{
  CGAL::Random_points_in_sphere_3<Point_3, PointCreator> gen(100.0);

  // generate 250 points randomly on a sphere of radius 100.0
  // and copy them to a vector
  std::vector<Point_3> points;
  CGAL::cpp0x::copy_n( gen, 250, std::back_inserter(points) );

  // define polyhedron to hold convex hull
  Polyhedron_3 poly;

  // compute convex hull of non-collinear points
  CGAL::convex_hull_3(points.begin(), points.end(), poly);

  std::cout << "The convex hull contains " << poly.size_of_vertices() <<
   " vertices" << std::endl;

  // assign a plane equation to each polyhedron facet using functor
   Plane_from_facet
  std::transform( poly.facets_begin(), poly.facets_end(),
   poly.planes_begin(),Plane_from_facet());


  return 0;
}
```

File: examples/Convex_hull_3/quickhull_3.cpp

CGAL::convex_hull_incremental_3

Function

Definition

The function *convex_hull_incremental_3* computes the convex hull polyhedron from a set of given three-dimensional points.

This function is provided for completeness and educational purposes. When an efficient incremental implementation is needed, using *CGAL::Delaunay_triangulation_3* together with *CGAL::convex_hull_3_to_polyhedron_3* is highly recommended.

#include <CGAL/convex_hull_incremental_3.h>

```
template <class InputIterator, class Polyhedron>
void convex_hull_incremental_3( InputIterator first,
                                InputIterator beyond,
                                Polyhedron& P,
                                bool test_correctness = false)
```

computes the convex hull polyhedron of the points in the range [*first*,*beyond*) and assigns it to *P*. If *test_correctness* is set to *true*, the tests described in [MNS+96] are used to determine the correctness of the resulting polyhedron.

Requirements

1. *Polyhedron* must provide a type *Polyhedron::Traits* that defines the following types
   - *Polyhedron::Traits::R*, which is a model of the representation class *R* required by *CGAL::Convex_hull_d_traits_3<R>*
   - *Polyhedron::Traits::Point*
2. *InputIterator::value_type* must be the same as *Polyhedron::Traits::Point*

See Also

CGAL::convex_hull_3 . . . . page 1408
CGAL::convex_hull_2 . . . . page 1379

Implementation

This function uses the d-dimensional convex hull incremental construction algorithm [CMS93] with d fixed to 3. The algorithm requires $O(n^2)$ time in the worst case and $O(n \log n)$ expected time.

See Also

CGAL::Convex_hull_d<R> . . . . page 1433

Example

The following example computes the convex hull of a set of 250 random points chosen uniformly in a sphere of radius 100.

```
#include <CGAL/Homogeneous.h>
#include <CGAL/point_generators_3.h>
#include <CGAL/algorithm.h>
#include <CGAL/Polyhedron_3.h>
#include <CGAL/convex_hull_incremental_3.h>
#include <vector>

#ifdef CGAL_USE_GMP
#include <CGAL/Gmpz.h>
typedef CGAL::Gmpz RT;
#else
#include <CGAL/MP_Float.h>
typedef CGAL::MP_Float RT;
#endif


typedef CGAL::Homogeneous<RT>                     K;
typedef K::Point_3                                Point_3;
typedef CGAL::Polyhedron_3< K>                    Polyhedron;
typedef CGAL::Creator_uniform_3<int, Point_3>    Creator;

int main()
{
  CGAL::Random_points_in_sphere_3<Point_3, Creator> gen(100.0);

  std::vector<Point_3> V;
  // generate 250 points randomly on a sphere of radius 100.0 and copy
  // them to a vector
  CGAL::cpp0x::copy_n( gen, 250, std::back_inserter(V) );

  Polyhedron P; // define polyhedron to hold convex hull

  // compute convex hull
  CGAL::convex_hull_incremental_3( V.begin(), V.end(), P, true);


  return 0;
}
```

File: examples/Convex_hull_3/incremental_hull_3.cpp

ConvexHullPolyhedronFacet_3

Definition

The requirements of the facet type of a polyhedron to be built by the function *convex_hull_3*.

Has Models

CGAL::Polyhedron_3<Traits>::Facet . . . . page 1807

Types

ConvexHullPolyhedronFacet_3:: Plane	plane equation type stored in facets.
ConvexHullPolyhedronFacet_3:: Halfedge_handle	handle to halfedge.
ConvexHullPolyhedronFacet_3:: Halfedge_around_facet_circulator	circulator of halfedges around a facet.

Creation

ConvexHullPolyhedronFacet_3 f;	default constructor.

Operations

Plane&	*f.plane()*	
const Plane&	*f.plane() const*	plane equation.
Halfedge_handle	*f.halfedge()*	an incident halfedge that points to *f*.
Halfedge_around_facet_circulator	*f.facet_begin()*	circulator of halfedges around the facet (counterclockwise).

See Also

CGAL::Polyhedron_3<Traits> . . . . page 1795
ConvexHullPolyhedronVertex_3 . . . . page 1415
ConvexHullPolyhedronHalfedge_3 . . . . page 1414

ConvexHullPolyhedronHalfedge_3

Definition

The requirements of the halfedge type required for the polyhedron built by the function *convex_hull_3*.

Has Models

CGAL::Polyhedron_3<Traits>::Halfedge . . . . page 1809

Creation

ConvexHullPolyhedronHalfedge_3 h;	default constructor.

Operations

Halfedge_handle	*h.opposite()*	the opposite halfedge.
Halfedge_handle	*h.next()*	the next halfedge around the facet.
Halfedge_handle	*h.prev()*	the previous halfedge around the facet.
bool	*h.is_border() const*	is true if *h* is a border halfedge.
Halfedge_around_facet_circulator		
	h.facet_begin()	circulator of halfedges around the facet (counterclockwise).
Vertex_handle	*h.vertex()*	the incident vertex of *h*.
Facet_handle	*h.facet()*	the incident facet of *h*. If *h* is a border halfedge the result is default construction of the handle.

See Also

CGAL::Polyhedron_3<Traits> . . . . page 1795
ConvexHullPolyhedronVertex_3 . . . . page 1415
ConvexHullPolyhedronFacet_3 . . . . page 1413

ConvexHullPolyhedronVertex_3

Definition

The requirements of the vertex type of the polyhedron to be built by the function *convex_hull_3*.

Has Models

CGAL::Polyhedron_3<Traits>::Vertex . . . . page 1812.

Creation

ConvexHullPolyhedronVertex_3 v;	default constructor.

Operations

Point&	*v.point()*	
const Point&	*v.point() const*	the point.

See Also

CGAL::Polyhedron_3<Traits> . . . . page 1795
ConvexHullPolyhedronFacet_3 . . . . page 1413
ConvexHullPolyhedronHalfedge_3 . . . . page 1414

Concept

ConvexHullPolyhedron_3

Definition

The requirements of the polyhedron type to be built by the function *convex_hull_3*.

Has Models

CGAL::Polyhedron_3<Traits> . . . . page 1795

Types

ConvexHullPolyhedron_3:: Point_3	type of point stored in a vertex
ConvexHullPolyhedron_3:: Vertex	a model of *ConvexHullPolyhedronVertex_3*
ConvexHullPolyhedron_3:: Halfedge	a model of *ConvexHullPolyhedronHalfedge_3*
ConvexHullPolyhedron_3:: Facet	a model of *ConvexHullPolyhedronFacet_3*
ConvexHullPolyhedron_3:: Halfedge_data_structure	halfedge data structure
ConvexHullPolyhedron_3:: Halfedge_handle	handle to halfedge
ConvexHullPolyhedron_3:: Halfedge_iterator	iterator for halfedge
ConvexHullPolyhedron_3:: Facet_handle	handle to facet
ConvexHullPolyhedron_3:: Facet_iterator	iterator for facet

Creation

Only a default constructor is required.

ConvexHullPolyhedron_3 p;

Operations

Facet_iterator	*p.facets_begin()*	iterator over all facets (excluding holes).
Facet_iterator	*p.facets_end()*	past-the-end iterator.
Halfedge_iterator	*p.P.halfedges_begin()*	iterator over all halfedges.
Halfedge_iterator	*p.P.halfedges_end()*	past-the-end iterator.
Halfedge_handle	*p.make_tetrahedron(Point_3 p1, Point_3 p2, Point_3 p3, Point_3 p4)*	adds a new tetrahedron to the polyhedral surface with its vertices initialized with *p1*, *p2*, *p3* and *p4*. Returns that halfedge of the tetrahedron which incident vertex is initialized with *p1*, the incident vertex of the next halfedge with *p2*, and the vertex thereafter with *p3*. The remaining fourth vertex is initialized with *p4*.

void *p.erase_facet(Halfedge_handle h)*

removes the incident facet of *h* and changes all halfedges incident to the facet into border edges or removes them from the polyhedral surface if they were already border edges.

Halfedge_handle *p.add_vertex_and_facet_to_border(Halfedge_handle h, Halfedge_handle g)*

creates a new facet within the hole incident to *h* and *g* by connecting the tip of *g* with the tip of *h* with two new halfedges and a new vertex and filling this separated part of the hole with a new facet, such that the new facet is incident to *g*. Returns the halfedge of the new edge that is incident to the new facet and the new vertex.

Halfedge_handle *p.add_facet_to_border(Halfedge_handle h, Halfedge_handle g)*

creates a new facet within the hole incident to *h* and *g* by connecting the tip of *g* with the tip of *h* with a new halfedge and filling this separated part of the hole with a new facet, such that the new facet is incident to *g*. Returns the halfedge of the new edge that is incident to the new facet.

Halfedge_handle *p.fill_hole(Halfedge_handle h)*

fills a hole with a newly created facet. Makes all border halfedges of the hole denoted by h incident to the new facet. Returns *h*.

void *p.delegate(Modifier_base<Halfedge_data_structure>& m)*

calls the *operator()* of the modifier *m*. See *Modifier_base* for a description of modifier design and its usage.

Concept

ConvexHullTraits_3

Definition

Requirements of the traits class to be used with the function *convex_hull_3*.

Types

ConvexHullTraits_3:: Point_3	The point type on which the convex hull algorithm operates
ConvexHullTraits_3:: Plane_3	a 3D plane
ConvexHullTraits_3:: Segment_3	a 3D segment
ConvexHullTraits_3:: Triangle_3	a 3D triangle
ConvexHullTraits_3:: Vector_3	a 3D vector
ConvexHullTraits_3:: Construct_plane_3	Function object type that provides *Plane_3 operator()(Point_3 p, Point_3 q, Point_3 r)*, which constructs and returns a plane passing through *p*, *q*, and *r* and oriented in a positive sense when seen from the positive side of the plane.
ConvexHullTraits_3:: Construct_segment_3	Function object type that provides *Segment_3 operator()(Point_3 p, Point_3 q)*, which constructs and returns the segment with source *p* and target *q*.
ConvexHullTraits_3:: Construct_triangle_3	Function object type that provides *Triangle_3 operator()(Point_3 p, Point_3 q, Point_3 r)*, which constructs and returns the triangle with vertices *p*, *q*, and *r*
ConvexHullTraits_3:: Construct_vector_3	Function object type that provides *Vector_3 operator()(Point_3 p, Point_3 q)*, which constructs and returns the vector *q*-*p*
ConvexHullTraits_3:: Equal_3	Predicate object type that provides *bool operator()(Point_3 p, Point_3 q)*, which determines if points *p* and *q* are equal or not
ConvexHullTraits_3:: Collinear_3	Predicate object type that provides *bool operator()(Point_3 p, Point_3 q, Point_3 r)*, which determines if points *p*, *q* and *r* are collinear or not
ConvexHullTraits_3:: Coplanar_3	Predicate object type that provides *bool operator()(Point_3 p, Point_3 q, Point_3 r, Point_3 s)*, which determines if points *p*, *q*, *r*, and *s* are coplanar or not
ConvexHullTraits_3:: Has_on_positive_side_3	Predicate object type that provides *bool operator()(Plane_3 h, Point_3 q)*, which determines of the point *q* is on the positive side of the halfspace *h*

ConvexHullTraits_3:: Less_distance_to_point_3	Predicate object type that provides a constructor taking a single *Point_3* object and *bool operator()(Point_3 q, Point_3 r)*, which returns true iff the distance from *q* to *p* is smaller than the distance from *r* to *p*, where *p* is the point passed to the object at construction.
ConvexHullTraits_3:: Less_signed_distance_to_plane_3	Predicate object type that provides *bool operator()(Plane_3 p, Point_3 q, Point_3 r)*, which returns true iff the signed distance from *q* to *p* is smaller than the signed distance from *r* to *p*

Creation

Only a copy constructor is required.

ConvexHullTraits_3 traits(& ch);

Operations

For each of the above function and predicate object types, *Func_obj_type*, a function must exist with the name *func_obj_type_object* that creates an instance of the function or predicate object type. For example:

Construct_plane_3 *traits.construct_plane_3_object()*

Has Models

CGAL::Convex_hull_traits_3<R> . . . . page 1420
All kernels of CGAL

Class

CGAL::Convex_hull_traits_3<R>

Definition

The class *Convex_hull_traits_3<R>* serves as a traits class for the function *convex_hull_3*. This is the default traits class for this function when *R* is a kernel with exact predicates but inexact constructions (note below that the type *Plane_3* is a triple of *Point_3* and not *R::Plane_3*).

#include <CGAL/Convex_hull_traits_3.h>

Is Model for the Concepts

ConvexHullTraits_3 . . . . page 1418
IsStronglyConvexTraits_3 . . . . page 1424

Types

typedef R::Point_3	*Point_3;*
typedef R::Segment_3	*Segment_3;*
typedef R::Triangle_3	*Triangle_3;*
Convex_hull_traits_3<R>:: Plane_3;	A triple of points, in order to avoid the need for exact constructions.
typedef R::Vector_3	*Vector_3;*
typedef Polyhedron_default_traits_3<R>	*Poly_traits;*
typedef Halfedge_data_structure_polyhedron_default_3<R>	*HDS;*
typedef Polyhedron_3<Poly_traits, HDS>	*Polyhedron_3;*
typedef R::Construct_segment_3	*Construct_segment_3;*
typedef R::Construct_ray_3	*Construct_ray_3;*
typedef R::Construct_plane_3	*Construct_plane_3;*
typedef R::Construct_vector_3	*Construct_vector_3;*
typedef R::Construct_triangle_3	*Construct_triangle_3;*
typedef R::Construct_centroid_3	*Construct_centroid_3;*
typedef R::Construct_orthogonal_vector_3	*Construct_orthogonal_vector_3;*
R::Equal_3	*Equal_3;*
R::Collinear_3	*Collinear_3;*
R::Coplanar_3	*Coplanar_3;*
R::Less_distance_to_point_3	*Less_distance_to_point_3;*
R::Has_on_positive_side_3	*Has_on_positive_side_3;*
R::Less_signed_dist_to_plane_3	*Less_signed_distance_to_plane_3;*
Projection_traits_xy_3<R>	*Traits_xy;*
Projection_traits_xz_3<R>	*Traits_xz;*
Projection_traits_yz_3<R>	*Traits_yz;*
R::Ray_3	*Ray_3;*
R::Has_on_3	*Has_on_3;*
R::Oriented_side_3	*Oriented_side_3;*
R::Do_intersect_3	*Do_intersect_3;*

Creation

Convex_hull_traits_3<R> traits(Convex_hull_traits_3& t); copy constructor.

Operations

Construct_segment_3	*traits.construct_segment_3_object() const*
Construct_ray_3	*traits.construct_ray_3_object() const*
Construct_plane_3	*traits.construct_plane_3_object() const*
Construct_triangle_3	*traits.construct_triangle_3_object() const*
Construct_vector_3	*traits.construct_vector_3_object() const*
Construct_centroid_3	*traits.construct_centroid_3_object() const*
Construct_orthogonal_vector_3	*traits.construct_orthogonal_vector_3_object() const*
Equal_3	*traits.equal_3_object() const*
Collinear_3	*traits.collinear_3_object() const*
Coplanar_3	*traits.coplanar_3_object() const*
Has_on_3	*traits.has_on_3_object() const*
Less_distance_to_point_3	*traits.less_distance_to_point_3_object() const*
Has_on_positive_side_3	*traits.has_on_positive_side_3_object() const*
Oriented_side_3	*traits.oriented_side_3_object() const*
Do_intersect_3	*traits.do_intersect_3_object() const*
Less_signed_distance_to_plane_3	*traits.less_signed_distance_to_plane_3_object() const*

See Also

CGAL::convex_hull_2 . . . . page 1379

Function

CGAL::convex_hull_3_to_polyhedron_3

Definition

The function *convex_hull_3_to_polyhedron_3* fills a polyhedron with the convex hull of a set of 3D points contained into a 3D triangulation of CGAL.

#include <CGAL/convex_hull_3_to_polyhedron_3.h>

template <class Triangulation_3, class Polyhedron_3>
void *convex_hull_3_to_polyhedron_3(Triangulation_3 T, Polyhedron_3& P)*

The polyhedron *P* is cleared and the convex hull of the set of 3D points is stored in *P*. The plane equations of each face are not computed.
Precondition: *T.dimension()*==3.

Requirements

This function requires the following:

1. *Triangulation_3* is a CGAL 3D triangulation.
2. *Polyhedron_3* is an instantiation of *CGAL::Polyhedron_3<Traits>*.

See Also

CGAL::convex_hull_3 . . . . page 1408

CGAL::is_strongly_convex_3

Function

Definition

The function *is_strongly_convex_3* determines if the vertices of a given polyhedron represents a strongly convex set of points or not. A set of points is said to be strongly convex if it consists of only extreme points (*i.e.*, vertices of the convex hull).

#include <CGAL/convexity_check_3.h>

template<class Polyhedron_3, class Traits>
bool *is_strongly_convex_3(Polyhedron_3& P, Traits traits = Default_traits)*

determines if the set of vertices of the polyhedron *P* represent a strongly convex set of points or not.
Precondition: The equations of the facet planes of the polyhedron must have already been computed.

The default traits class is the kernel in which the type *Polyhedron_3::Point_3* is defined.

Requirements

1. *Polyhedron_3::Point_3* is equivalent to *Traits::Point_3*.
2. *Traits* is a model of the concept *IsStronlyConvexTraits_3*
3. *Polyhedron_3* must define the following types:
   - *Polyhedron_3::Facet_iterator*
   - *Polyhedron_3::Vertex_iterator*

   and the following member functions:

   - *facets_begin()*
   - *facets_end()*
   - *vertices_begin()*
   - *vertices_end()*

   The vertex type of *Polyhedron_3* must be a model of *ConvexHullPolyhedronVertex_3*; the facet type must be *ConvexHullPolyhedronFacet_3*.

See Also

CGAL::is_ccw_strongly_convex_2 . . . . page 1391
CGAL::is_cw_strongly_convex_2 . . . . page 1392

Implementation

This function implements the tests described in [MNS+96] to determine convexity and requires $O(e+f)$ time for a polyhedron with e edges and f faces.

Concept

IsStronglyConvexTraits_3

Definition

Requirements of the traits class used by the function *is_strongly_convex_3*, which is used for postcondition checking by *convex_hull_3*.

Types

IsStronglyConvexTraits_3:: Plane_3	a 3D plane
IsStronglyConvexTraits_3:: Point_3	a 3D point
IsStronglyConvexTraits_3:: Ray_3	a 3D ray
IsStronglyConvexTraits_3:: Triangle_3	a 3D triangle
IsStronglyConvexTraits_3:: Construct_ray_3	Function object type that provides *Ray_3 operator()(Point_3 p, Point_3 q)*, which constructs and returns the ray based at point *p* and passing though *q*
IsStronglyConvexTraits_3:: Construct_triangle_3	Function object type that provides *Triangle_3 operator()(Point_3 p, Point_3 q, Point_3 r)*, which constructs and returns the triangle with vertices *p*, *q*, and *r*
IsStronglyConvexTraits_3:: Coplanar_3	Predicate object type that provides *bool operator()(Point_3 p, Point_3 q, Point_3 r, Point_3 s)*, which determines if points *p*, *q*, *r*, and *s* are coplanar or not
IsStronglyConvexTraits_3:: Do_intersect_3	Function object type that provides *CGAL::Object operator()(Triangle_3 t, Ray_3 r)*, which returns *true* iff *t* and *r* intersect.
IsStronglyConvexTraits_3:: Has_on_positive_side_3	Predicate object type that provides *bool operator()(Plane_3 h, Point_3 q)*, which determines of the point *q* is on the positive side of the halfspace *h*
IsStronglyConvexTraits_3:: Oriented_side_3	Predicate object type that provides *Oriented_side operator()(Plane_3 p, Point_3 q)*, which determines the position of point *q* relative to plane *p*

Creation

Only a copy constructor is required.

IsStronglyConvexTraits_3 traits(& t);

Operations

For each of the above function and predicate object types, *Func_obj_type*, a function must exist with the name *func_obj_type_object* that creates an instance of the function or predicate object type. For example:

Construct_ray_3 *traits.construct_ray_3_object()*

Has Models

CGAL::Convex_hull_traits_3<R> . . . . page 1420
All kernels of CGAL

See Also

ConvexHullTraits_3 . . . . page 1418
CGAL::is_strongly_convex_3 . . . . page 1423

Chapter 17

dD Convex Hulls and Delaunay Triangulations

Susan Hert and Michael Seel

Contents

17.1 Introduction

A subset $S \subseteq \mathbb{R}^d$ is convex if for any two points p and q in the set the line segment with endpoints p and q is contained in S. The convex hull of a set S is the smallest convex set containing S. The convex hull of a set of points P is a convex polytope with vertices in P. A point in P is an extreme point (with respect to P) if it is a vertex of the convex hull of P. A set of points is said to be strongly convex if it consist of only extreme points.

This chapter describes the class provided in CGAL for producing convex hull in arbitrary dimensions. There is an intimate relationship between the Delaunay triangulation of a point set S and the convex hull of *lift(S)*: The nearest site Delaunay triangulation is the projection of the lower hull and the furthest site Delaunay triangulation is the upper hull. Here we also describe the companion class to the convex hull class that computes nearest and furthest site Delaunay triangulations.

17.2 dD Convex Hull

The class *CGAL::Convex_hull_d<R>* is used to represent the convex hull of a set of points in d-dimensional space. This class supports incremental construction of hulls, and provides a rich interface for exploration. There are also output routines for hulls of dimension 2 and 3.

The convex hull class is parameterized by a traits class that provides d-dimensional data types and predicates. The class *Convex_hull_d_traits_3* adapts any low-dimensional standard kernel model *e.g.*, *Homogeneous<RT>* or *Cartesian<FT>* for use with *Convex_hull_d*, where the dimension is fixed to three. The validity of the computed convex hull can be checked using the member function *is_valid*, which implements the algorithm of Mehlhorn *et al.*[MNS+96] to determine if the vertices of a given polytope constitute a strongly convex point set or not.

The implementation follows the papers [CMS93] and [BMS94].

17.3 Delaunay Triangulation

There is a class type with a thorough interface providing the construction and exploration of closest and furthest site Delaunay simplicial complexes in arbitrary higher dimension. The class *CGAL::Delaunay_d<R, Lifted_R>* provides an implementation via the lifting map to higher dimensional convex hulls.. The class supports incremental construction of Delaunay triangulations and various kind of query operations (in particular, nearest and furthest neighbor queries and range queries with spheres and simplices).

dD Convex Hulls and Delaunay Triangulations Reference Manual

Susan Hert and Michael Seel

A subset $S \subseteq \mathbb{R}^3$ is convex if for any two points p and q in the set the line segment with endpoints p and q is contained in S. The convex hull of a set S is the smallest convex set containing S. The convex hull of a set of points P is a convex polytope with vertices in P. A point in P is an extreme point (with respect to P) if it is a vertex of the convex hull of P.

CGAL provides functions for computing convex hulls in two, three and arbitrary dimensions as well as functions for testing if a given set of points in is strongly convex or not. This chapter describes the class available for arbitrary dimensions and its companion class for computing the nearest and furthest side Delaunay triangulation.

17.4 Classified Reference Pages

Concepts

Classes

17.5 Alphabetical List of Reference Pages

ConvexHullTraits_d

Definition

Requirements of the traits class to be used with the class *Convex_hull_d*.

Types

ConvexHullTraits_d:: Point_d	the dD point type on which the convex hull algorithm operates
ConvexHullTraits_d:: Hyperplane_d	a dD plane
ConvexHullTraits_d:: Vector_d	a dD vector
ConvexHullTraits_d:: Ray_d	a dD ray
ConvexHullTraits_d:: RT	an arithmetic ring type
ConvexHullTraits_d:: Construct_vector_d	Function object type that provides *Vector_d operator()(int d, CGAL::Null_vector)*, which constructs and returns the null vector.
ConvexHullTraits_d:: Construct_hyperplane_d	Function object type that provides *Hyperplane_d operator()(ForwardIterator first, ForwardIterator last, Point_d p, CGAL::Oriented_side side)*, which constructs and returns a hyperplane passing through the points in *tuple[first,last)* and oriented such that *p* is on the side *side* of the returned hyperplane. When *side==ON_ORIENTED_BOUNDARY* then any hyperplane containing the tuple is returned.
ConvexHullTraits_d:: Vector_to_point_d	Function object type that provides *Point_d operator()(Vector_d v)*, which constructs and returns the point defined by $0+v$.
ConvexHullTraits_d:: Point_to_vector_d	Function object type that provides *Vector_d operator()(Point_d v)*, which constructs and returns the vector defined by $p-0$.
ConvexHullTraits_d:: Orientation_d	Function object type that provides *Orientation operator()(ForwardIterator first, ForwardIterator last)*, which determines the orientation of the points *tuple[first,last)*.
ConvexHullTraits_d:: Orthogonal_vector_d	Function object type that provides *Vector_d operator()(Hyperplane_d h)*, which constructs and returns a vector orthogonal to h and pointing from the boundary into its positive halfspace.
ConvexHullTraits_d:: Oriented_side_d	Predicate object type that provides *Oriented_side operator()(Hyperplane_d h, Point_d p)*, which determines the oriented side of p with respect to h.

ConvexHullTraits_d:: Has_on_positive_side_d	Predicate object type that provides *bool operator()(Hyperplane_d h, Point_d p)*, which return true iff *p* lies in the positive halfspace determined by *h*.
ConvexHullTraits_d:: Affinely_independent_d	Predicate object type that provides *bool operator()(ForwardIterator first, ForwardIterator last)*, which determines if the points *tuple[first,last)* are affinely independent.
ConvexHullTraits_d:: Contained_in_simplex_d	Predicate object type that provides *bool operator()(ForwardIterator first, ForwardIterator last, Point_d p)*, which determines if *p* is contained in the closed simplex defined by the points in *tuple[first,last)*.
ConvexHullTraits_d:: Contained_in_affined_hull_d	
	Predicate object type that provides *bool operator()(ForwardIterator first, ForwardIterator last, Point_d p)*, which determines if *p* is contained in the affine hull of the points in *tuple[first,last)*.
ConvexHullTraits_d:: Intersect_d	Predicate object type that provides *Object operator()(Ray_d r, Hyperplane_d h)*, which determines if *r* and *h* intersect and returns the corresponding polymorphic object.

Creation

A default constructor and copy constructor is required.

Operations

For each of the above function and predicate object types, *Func_obj_type*, a function must exist with the name *func_obj_type_object* that creates an instance of the function or predicate object type. For example:

Construct_vector_d *traits.construct_vector_d_object()*

Has Models

CGAL::Cartesian_d<FT,LA>
CGAL::Homogeneous_d<RT,LA>
CGAL::Convex_hull_d_traits_3<R> . . . . page 1440

CGAL::Convex_hull_d<R>

Definition

An instance *C* of type *Convex_hull_d<R>* is the convex hull of a multi-set *S* of points in *d*-dimensional space. We call *S* the underlying point set and *d* or *dim* the dimension of the underlying space. We use *dcur* to denote the affine dimension of *S*. The data type supports incremental construction of hulls.

The closure of the hull is maintained as a simplicial complex, i.e., as a collection of simplices. The intersection of any two is a face of both[1]. In the sequel we reserve the word simplex for the simplices of dimension *dcur*. For each simplex there is a handle of type *Simplex_handle* and for each vertex there is a handle of type *Vertex_handle*. Each simplex has $1+dcur$ vertices indexed from 0 to *dcur*; for a simplex *s* and an index *i*, *C.vertex(s,i)* returns the *i*-th vertex of *s*. For any simplex *s* and any index *i* of *s* there may or may not be a simplex *t* opposite to the *i*-th vertex of *s*. The function *C.opposite_simplex(s,i)* returns *t* if it exists and returns *Simplex_handle()* (the undefined handle) otherwise. If *t* exists then *s* and *t* share *dcur* vertices, namely all but the vertex with index *i* of *s* and the vertex with index *C.index_of_vertex_in_opposite_simplex(s,i)* of *t*. Assume that *t* exists and let *j* = *C.index_of_vertex_in_opposite_simplex(s,i)*. Then *s* = *C.opposite_simplex(t,j)* and *i* = *C.index_of_vertex_in_opposite_simplex(t,j)*.

The boundary of the hull is also a simplicial complex. All simplices in this complex have dimension $dcur-1$. For each boundary simplex there is a handle of type *Facet_handle*. Each facet has *dcur* vertices indexed from 0 to $dcur-1$. If *dcur* > *1* then for each facet *f* and each index *i*, $0 \leq i < dcur$, there is a facet *g* opposite to the *i*-th vertex of *f*. The function *C.opposite_facet(f,i)* returns *g*. Two neighboring facets *f* and *g* share *dcur - 1* vertices, namely all but the vertex with index *i* of *f* and the vertex with index *C.index_of_vertex_in_opposite_facet(f,i)* of *g*. Let *j* = *C.index_of_vertex_in_opposite_facet(f,i)*. Then *f* = *C.opposite_facet(g,j)* and *i* = *C.index_of_vertex_in_opposite_facet(g,j)*.

Types

Convex_hull_d<R>:: R	the representation class.
Convex_hull_d<R>:: Point_d	the point type.
Convex_hull_d<R>:: Hyperplane_d	the hyperplane type.
Convex_hull_d<R>:: Simplex_handle	handle for simplices.
Convex_hull_d<R>:: Facet_handle	handle for facets.
Convex_hull_d<R>:: Vertex_handle	handle for vertices.
Convex_hull_d<R>:: Simplex_iterator	iterator for simplices.
Convex_hull_d<R>:: Facet_iterator	iterator for facets.
Convex_hull_d<R>:: Vertex_iterator	iterator for vertices.

[1] The empty set if a facet of every simplex.

Convex_hull_d<R>:: Hull_vertex_iterator iterator for vertices that are part of the convex hull.

Note that each iterator fits the handle concept, i.e. iterators can be used as handles. Note also that all iterator and handle types come also in a const flavor, e.g., *Vertex_const_iterator* is the constant version of *Vertex_iterator*. Const correctness requires to use the const version whenever the convex hull object is referenced as constant. The *Hull_vertex_iterator* is convertible to *Vertex_iterator* and *Vertex_handle*.

Convex_hull_d<R>:: Point_const_iterator const iterator for all inserted points.

Convex_hull_d<R>:: Hull_point_const_iterator

const iterator for all points that are part of the convex hull.

Creation

Convex_hull_d<R> C(int d, R Kernel = R());

creates an instance C of type *Convex_hull_d*. The dimension of the underlying space is d and S is initialized to the empty point set. The traits class R specifies the models of all types and the implementations of all geometric primitives used by the convex hull class. The default model is one of the d-dimensional representation classes (e.g., *Homogeneous_d*).

The data type *Convex_hull_d* offers neither copy constructor nor assignment operator.

Requirements

R is a model of the concept *ConvexHullTraits_d* .

Operations

All operations below that take a point x as argument have the common precondition that x is a point of ambient space.

int *C.dimension()*

returns the dimension of ambient space

int *C.current_dimension()*

returns the affine dimension $dcur$ of S.

Point_d *C.associated_point(Vertex_handle v)*

returns the point associated with vertex v.

Vertex_handle *C.vertex_of_simplex(Simplex_handle s, int i)*

returns the vertex corresponding to the i-th vertex of s.

Precondition: $0 \leq i \leq dcur$.

Point_d *C.point_of_simplex(Simplex_handle s, int i)*

same as *C.associated_point(C.vertex_of_simplex(s,i))*.

Simplex_handle *C.opposite_simplex(Simplex_handle s, int i)*

returns the simplex opposite to the i-th vertex of s (*Simplex_handle()* if there is no such simplex).
Precondition: $0 \leq i \leq dcur$.

int *C.index_of_vertex_in_opposite_simplex(Simplex_handle s, int i)*

returns the index of the vertex opposite to the i-th vertex of s.
Precondition: $0 \leq i \leq dcur$ and there is a simplex opposite to the i-th vertex of s.

Simplex_handle *C.simplex(Vertex_handle v)*

returns a simplex of which v is a node. Note that this simplex is not unique.

int *C.index(Vertex_handle v)*

returns the index of v in *simplex(v)*.

Vertex_handle *C.vertex_of_facet(Facet_handle f, int i)*

returns the vertex corresponding to the i-th vertex of f.
Precondition: $0 \leq i < dcur$.

Point_d *C.point_of_facet(Facet_handle f, int i)*

same as *C.associated_point(C.vertex_of_facet(f,i))*.

Facet_handle *C.opposite_facet(Facet_handle f, int i)*

returns the facet opposite to the i-th vertex of f (*Facet_handle()* if there is no such facet).
Precondition: $0 \leq i < dcur$ and $dcur > 1$.

int *C.index_of_vertex_in_opposite_facet(Facet_handle f, int i)*

returns the index of the vertex opposite to the i-th vertex of f.
Precondition: $0 \leq i < dcur$ and *dcur > 1*.

Hyperplane_d *C.hyperplane_supporting(Facet_handle f)*

returns a hyperplane supporting facet f. The hyperplane is oriented such that the interior of C is on the negative side of it.
Precondition: f is a facet of C and *dcur > 1*.

Vertex_handle *C.insert(Point_d x)*

adds point x to the underlying set of points. If x is equal to (the point associated with) a vertex of the current hull this vertex is returned and its associated point is changed to x. If x lies outside the current hull, a new vertex v with associated point x is added to the hull and returned. In all other cases, i.e., if x lies in the interior of the hull or on the boundary but not on a vertex, the current hull is not changed and *Vertex_handle()* is returned. If *CGAL_CHECK_EXPENSIVE* is defined then the validity check *is_valid(true)* is executed as a post condition.

template <typename Forward_iterator>
void *C.insert(Forward_iterator first, Forward_iterator last)*

adds S = *set [first,last)* to the underlying set of points. If any point *S[i]* is equal to (the point associated with) a vertex of the current hull its associated point is changed to *S[i]*.

bool *C.is_dimension_jump(Point_d x)*

returns true if x is not contained in the affine hull of S.

std::list<Facet_handle>

C.facets_visible_from(Point_d x)

returns the list of all facets that are visible from x.

Precondition: x is contained in the affine hull of S.

Bounded_side *C.bounded_side(Point_d x)*

returns *ON_BOUNDED_SIDE* (*ON_BOUNDARY*,*ON_UNBOUNDED_SIDE*) if x is contained in the interior (lies on the boundary, is contained in the exterior) of C.
Precondition: x is contained in the affine hull of S.

void *C.clear(int d)*

re-initializes C to an empty hull in d-dimensional space.

int *C.number_of_vertices()*

returns the number of vertices of C.

int *C.number_of_facets()*

returns the number of facets of C.

int *C.number_of_simplices()*

returns the number of bounded simplices of C.

void *C.print_statistics()*

gives information about the size of the current hull and the number of visibility tests performed.

bool *C.is_valid(bool throw_exceptions = false)*

checks the validity of the data structure. If *throw_exceptions* == *thrue* then the program throws the following exceptions to inform about the problem.
chull_has_center_on_wrong_side_of_hull_facet the hyperplane supporting a facet has the wrong orientation.
chull_has_local_non_convexity a ridge is locally non convex.
chull_has_double_coverage the hull has a winding number larger than 1.

Lists and Iterators

Vertex_iterator	*C.vertices_begin()*	an iterator of C to start the iteration over all vertices of the complex.
Vertex_iterator	*C.vertices_end()*	the past the end iterator for vertices.
Simplex_iterator	*C.simplices_begin()*	an iterator of C to start the iteration over all simplices of the complex.
Simplex_iterator	*C.simplices_end()*	the past the end iterator for simplices.
Facet_iterator	*C.facets_begin()*	an iterator of C to start the iteration over all facets of the complex.

Facet_iterator	*C.facets_end()*	the past the end iterator for facets.
Hull_vertex_iterator	*C.hull_vertices_begin()*	an iterator to start the iteration over all vertices of C that are part of the convex hull.
Hull_vertex_iterator	*C.hull_vertices_end()*	the past the end iterator for hull vertices.
Point_const_iterator	*C.points_begin()*	returns the start iterator for all points that have been inserted to construct C.
Point_const_iterator	*C.points_end()*	returns the past the end iterator for points.
Hull_point_const_iterator	*C.hull_points_begin()*	returns an iterator to start the iteration over all points in the convex hull C. Included are points in the interior of facets.
Hull_point_const_iterator	*C.hull_points_end()*	returns the past the end iterator for points in the convex hull.
template <typename Visitor> *void*	*C.visit_all_facets(Visitor V)*	each facet of C is visited by the visitor object V. V has to have a function call operator: *void operator()(Facet_handle) const*
std::list<Point_d>	*C.all_points()*	returns a list of all points that have been inserted to construct C.
std::list<Vertex_handle>	*C.all_vertices()*	returns a list of all vertices of C (also interior ones).
std::list<Simplex_handle>	*C.all_simplices()*	returns a list of all simplices in C.
std::list<Facet_handle>	*C.all_facets()*	returns a list of all facets of C.

Iteration Statements

forall_ch_vertices(v, C) { "the vertices of C are successively assigned to v" }

forall_ch_simplices(s, C) { "the simplices of C are successively assigned to s" }

forall_ch_facets(f, C) { "the facets of C are successively assigned to f" }

Implementation

The implementation of type *Convex_hull_d* is based on [CMS93] and [BMS94]. The details of the implementation can be found in the implementation document available at the download site of this package.

The time and space requirements are input dependent. Let C_1, C_2, C_3, ... be the sequence of hulls constructed and for a point x let k_i be the number of facets of C_i that are visible from x and that are not already facets of C_{i-1}. Then the time for inserting x is $O(dim \sum_i k_i)$ and the number of new simplices constructed during the insertion of x is the number of facets of the hull which were not already facets of the hull before the insertion.

The data type *Convex_hull_d* is derived from *Regular_complex_d*. The space requirement of regular complexes is essentially $12(dim+2)$ bytes times the number of simplices plus the space for the points. *Convex_hull_d* needs an additional $8+(4+x)dim$ bytes per simplex where x is the space requirement of the underlying number type and an additional 12 bytes per point. The total is therefore $(16+x)dim+32$ bytes times the number of simplices plus $28+x \cdot dim$ bytes times the number of points.

Low Dimensional Conversion Routine

include <*CGAL/Convex_hull_d_to_polyhedron_3.h*>

template <class R, class T, class HDS>
void *convex_hull_d_to_polyhedron_3(C, Polyhedron_3<T,HDS>& P)*

converts the convex hull C to polyhedral surface stored in P.

Precondition: *dim == 3* and *dcur == 3*.

Low Dimensional Output Routines

template <class R>
void *d3_surface_map(C, GRAPH< typename ::Point_d ,int>& G)*

constructs the representation of the surface of C as a bidirected LEDA graph G.

Precondition: *dim == 3*.

Class

CGAL::Convex_hull_d_traits_3<R>

Definition

The class *Convex_hull_d_traits_3<R>* serves as a traits class for the class *Convex_hull_d*. This is a traits class that adapts any low-dimensional standard kernel model, e.g. *Homogeneous<RT>* or *Cartesian<FT>* for the fixed 3-dimensional usage of *Convex_hull_d*.

#include <CGAL/Convex_hull_d_traits_3.h>

Is Model for the Concepts

ConvexHullTraits_d

Creation

Convex_hull_d_traits_3<R> traits;	default constructor.

DelaunayLiftedTraits_d

Definition

Requirements of the second traits class to be used with the class *Delaunay_d*.

Types

DelaunayLiftedTraits_d:: Point_d	the dD point type on which the Delaunay algorithm operates
DelaunayLiftedTraits_d:: Hyperplane_d	a dD plane
DelaunayLiftedTraits_d:: Vector_d	a dD vector
DelaunayLiftedTraits_d:: Ray_d	a dD ray
DelaunayLiftedTraits_d:: RT	a arithmetic ring type
DelaunayLiftedTraits_d:: Construct_vector_d	Function object type that provides *Vector_d operator()(int d, CGAL::Null_vector)*, which constructs and returns the null vector.
DelaunayLiftedTraits_d:: Construct_hyperplane_d	Function object type that provides *Hyperplane_d operator()(ForwardIterator first, ForwardIterator last, Point_d p, CGAL::Oriented_side side)*, which constructs and returns a hyperplane passing through the points in *tuple[first,last)* and oriented such that *p* is on the side *side* of the returned hyperplane. When *side==ON_ORIENTED_BOUNDARY* then any hyperplane containing the tuple is returned.
DelaunayLiftedTraits_d:: Vector_to_point_d	Function object type that provides *Point_d operator()(Vector_d v)*, which constructs and returns the point defined by $0+v$.
DelaunayLiftedTraits_d:: Point_to_vector_d	Function object type that provides *Vector_d operator()(Point_d v)*, which constructs and returns the vector defined by $p-0$.
DelaunayLiftedTraits_d:: Orientation_d	Function object type that provides *Orientation operator()(ForwardIterator first, ForwardIterator last)*, which determines the orientation of the points *tuple[first,last)*.
DelaunayLiftedTraits_d:: Orthogonal_vector_d	Function object type that provides *Vector_d operator()(Hyperplane_d h)*, which constructs and returns a vector orthogonal to *h* and pointing from the boundary into its positive halfspace.
DelaunayLiftedTraits_d:: Oriented_side_d	Predicate object type that provides *Oriented_side operator()(Hyperplane_d h, Point_d p)*, which determines the oriented side of *p* with respect to *h*.

DelaunayLiftedTraits_d:: Has_on_positive_side_d — Predicate object type that provides *bool operator()(Hyperplane_d h, Point_d p)*, which return true iff p lies in the positive halfspace determined by h.

DelaunayLiftedTraits_d:: Affinely_independent_d — Predicate object type that provides *bool operator()(ForwardIterator first, ForwardIterator last)*, which determines if the points *tuple[first,last)* are affinely independent.

DelaunayLiftedTraits_d:: Contained_in_simplex_d — Predicate object type that provides *bool operator()(ForwardIterator first, ForwardIterator last, Point_d p)*, which determines if p is contained in the closed simplex defined by the points in *tuple[first,last)*.

DelaunayLiftedTraits_d:: Contained_in_affined_hull_d

Predicate object type that provides *bool operator()(ForwardIterator first, ForwardIterator last, Point_d p)*, which determines if p is contained in the affine hull of the points in *tuple[first,last)*.

DelaunayLiftedTraits_d:: Intersect_d — Predicate object type that provides *Object operator()(Ray_d r, Hyperplane_d h)*, which determines if r and h intersect and returns the corresponding polymorphic object.

The previous requirements are all identical to the concept *ConvexHullTraits_d*. The Delaunay class adds the following requirements.

DelaunayLiftedTraits_d:: Project_along_d_axis_d — Predicate object type that provides *DelaunayTraits_d::Point_d operator()(Point_d p)*, which determines the $d-1$-dimensional point from the d-dimensional point p while discarding the last coordinate.

DelaunayLiftedTraits_d:: Lift_to_paraboloid_d — Predicate object type that provides *Point_d operator()(DelaunayTraits_d::Point_d p)*, which determines the d-dimensional point from the $d-1$-dimensional point p while lifting it to the paraboloid of revolution.

DelaunayLiftedTraits_d:: Component_accessor_d — Predicate object type that provides *RT homogeneous(Vector_d v,int i)* and *int dimension(Vector_d v)*, where the former determines the ith coordinate of v and the latter the dimension of v.

Creation

A default constructor and copy constructor is required.

Operations

For each of the above function and predicate object types, *Func_obj_type*, a function must exist with the name *func_obj_type_object* that creates an instance of the function or predicate object type. For example:

Construct_vector_d *traits.construct_vector_d_object()*

Has Models

CGAL::Cartesian_d<FT,LA>
CGAL::Homogeneous_d<RT,LA>

Concept

DelaunayTraits_d

Definition

Requirements of the first traits class to be used with the class *Delaunay_d*.

Types

DelaunayTraits_d:: Point_d	the dD point type on which the Delaunay algorithm operates
DelaunayTraits_d:: Sphere_d	a dD sphere
DelaunayTraits_d:: FT	an arithmetic field type
DelaunayTraits_d:: Point_of_sphere_d	Predicate object type that provides *Point_d operator()(Sphere_d s, int i)*, which returns the *i*th point defining sphere *s*.
DelaunayTraits_d:: Construct_sphere_d	Predicate object type that provides *Sphere_d operator()(int d, ForwardIterator first, ForwardIterator last)*, which returns a dD sphere through the points in *tuple[first,last)*.
DelaunayTraits_d:: Contained_in_simplex_d	Predicate object type that provides *bool operator()(ForwardIterator first, ForwardIterator last, Point_d p)*, which determines if *p* is contained in the closed simplex defined by the points in *tuple[first,last)*.
DelaunayTraits_d:: Squared_distance_d	Predicate object type that provides *FT operator()(Point_d p,Point_d q)*, which determines the squared distance from *p* to *q*.
DelaunayTraits_d:: Affinely_independent_d	Predicate object type that provides *bool operator()(ForwardIterator first, ForwardIterator last)*, which determines if the points in *tuple[first,last)* are affinely independent.

Creation

A default constructor and copy constructor is required.

Operations

For each of the above function and predicate object types, *Func_obj_type*, a function must exist with the name *func_obj_type_object* that creates an instance of the function or predicate object type. For example:

Construct_sphere_d	*traits.construct_sphere_d_object()*

Has Models

CGAL::Cartesian_d<FT,LA>
CGAL::Homogeneous_d<RT,LA>

Class

CGAL::Delaunay_d< R, Lifted_R >

Definition

An instance *DT* of type *Delaunay_d< R, Lifted_R >* is the nearest and furthest site Delaunay triangulation of a set *S* of points in some *d*-dimensional space. We call *S* the underlying point set and *d* or *dim* the dimension of the underlying space. We use *dcur* to denote the affine dimension of *S*. The data type supports incremental construction of Delaunay triangulations and various kind of query operations (in particular, nearest and furthest neighbor queries and range queries with spheres and simplices).

A Delaunay triangulation is a simplicial complex. All simplices in the Delaunay triangulation have dimension *dcur*. In the nearest site Delaunay triangulation the circumsphere of any simplex in the triangulation contains no point of *S* in its interior. In the furthest site Delaunay triangulation the circumsphere of any simplex contains no point of *S* in its exterior. If the points in *S* are co-circular then any triangulation of *S* is a nearest as well as a furthest site Delaunay triangulation of *S*. If the points in *S* are not co-circular then no simplex can be a simplex of both triangulations. Accordingly, we view *DT* as either one or two collection(s) of simplices. If the points in *S* are co-circular there is just one collection: the set of simplices of some triangulation. If the points in *S* are not co-circular there are two collections. One collection consists of the simplices of a nearest site Delaunay triangulation and the other collection consists of the simplices of a furthest site Delaunay triangulation.

For each simplex of maximal dimension there is a handle of type *Simplex_handle* and for each vertex of the triangulation there is a handle of type *Vertex_handle*. Each simplex has *1 + dcur* vertices indexed from 0 to *dcur*. For any simplex *s* and any index *i*, *DT.vertex_of(s,i)* returns the *i*-th vertex of *s*. There may or may not be a simplex *t* opposite to the vertex of *s* with index *i*. The function *DT.opposite_simplex(s,i)* returns *t* if it exists and returns *Simplex_handle()* otherwise. If *t* exists then *s* and *t* share *dcur* vertices, namely all but the vertex with index *i* of *s* and the vertex with index *DT.index_of_vertex_in_opposite_simplex(s,i)* of *t*. Assume that $t = DT.opposite\_simplex(s,i)$ exists and let $j = DT.index\_of\_vertex\_in\_opposite\_simplex(s,i)$. Then *s* = *DT.opposite_simplex(t,j)* and *i* = *DT.index_of_vertex_in_opposite_simplex(t,j)*. In general, a vertex belongs to many simplices.

Any simplex of *DT* belongs either to the nearest or to the furthest site Delaunay triangulation or both. The test *DT.simplex_of_nearest(dt_simplex s)* returns true if *s* belongs to the nearest site triangulation and the test *DT.simplex_of_furthest(dt_simplex s)* returns true if *s* belongs to the furthest site triangulation.

Inherits From

Convex_hull_d<Lifted_R>

Types

Delaunay_d< R, Lifted_R >:: Simplex_handle	handles to the simplices of the complex.
Delaunay_d< R, Lifted_R >:: Vertex_handle	handles to vertices of the complex.
Delaunay_d< R, Lifted_R >:: Point_d	the point type
Delaunay_d< R, Lifted_R >:: Sphere_d	the sphere type

enum Delaunay_voronoi_kind { *NEAREST, FURTHEST*};

interface flags

To use these types you can typedef them into the global scope after instantiation of the class. We use *Vertex_handle* instead of *Delaunay_d< R, Lifted_R >::Vertex_handle* from now on. Similarly we use *Simplex_handle*.

Delaunay_d< R, Lifted_R >:: Point_const_iterator	the iterator for points.
Delaunay_d< R, Lifted_R >:: Vertex_iterator	the iterator for vertices.
Delaunay_d< R, Lifted_R >:: Simplex_iterator	the iterator for simplices.

Creation

Delaunay_d< R, Lifted_R > DT(int d, R k1 = R(), Lifted_R k2 = Lifted_R());

creates an instance *DT* of type *Delaunay_d*. The dimension of the underlying space is *d* and *S* is initialized to the empty point set. The traits class *R* specifies the models of all types and the implementations of all geometric primitives used by the Delaunay class. The traits class *Lifted_R* specifies the models of all types and the implementations of all geometric primitives used by the base class of *Delaunay_d< R, Lifted_R >*. The second template parameter defaults to the first: *Delaunay_d<R> = Delaunay_d<R, Lifted_R = R >*.

The data type *Delaunay_d* offers neither copy constructor nor assignment operator.

Requirements

R is a model of the concept *DelaunayTraits_d* . *Lifted_R* is a model of the concept *DelaunayLiftedTraits_d* .

Operations

All operations below that take a point x as an argument have the common precondition that $x.dimension() = DT.dimension()$.

int *DT.dimension()* returns the dimension of ambient space

int *DT.current_dimension()*

returns the affine dimension of the current point set, i.e., -1 is *S* is empty, 0 if *S* consists of a single point, 1 if all points of *S* lie on a common line, etc.

bool *DT.is_simplex_of_nearest(Simplex_handle s)*

returns true if *s* is a simplex of the nearest site triangulation.

bool *DT.is_simplex_of_furthest(Simplex_handle s)*

returns true if *s* is a simplex of the furthest site triangulation.

Vertex_handle *DT.vertex_of_simplex(Simplex_handle s, int i)*

returns the vertex associated with the *i*-th node of *s*.
Precondition: $0 \leq i \leq dcur$.

Point_d *DT.associated_point(Vertex_handle v)*

returns the point associated with vertex *v*.

Point_d *DT.point_of_simplex(Simplex_handle s, int i)*

returns the point associated with the *i*-th vertex of *s*.
Precondition: $0 \leq i \leq dcur$.

Simplex_handle *DT.opposite_simplex(Simplex_handle s, int i)*

returns the simplex opposite to the *i*-th vertex of *s* (*Simplex_handle()* if there is no such simplex).
Precondition: $0 \leq i \leq dcur$.

int *DT.index_of_vertex_in_opposite_simplex(Simplex_handle s, int i)*

returns the index of the vertex opposite to the *i*-th vertex of *s*.
Precondition: $0 \leq i \leq dcur$.

Simplex_handle *DT.simplex(Vertex_handle v)*

returns a simplex of the nearest site triangulation incident to *v*.

int *DT.index(Vertex_handle v)*

returns the index of *v* in *DT.simplex(v)*.

bool *DT.contains(Simplex_handle s, Point_d x)*

returns true if *x* is contained in the closure of simplex *s*.

bool *DT.empty()* decides whether *DT* is empty.

void *DT.clear()* re-initializes *DT* to the empty Delaunay triangulation.

Vertex_handle *DT.insert(Point_d x)*

inserts point x into DT and returns the corresponding *Vertex_handle*. More precisely, if there is already a vertex v in DT positioned at x (i.e., *associated_point(v)* is equal to x) then *associated_point(v)* is changed to x (i.e., *associated_point(v)* is made identical to x) and if there is no such vertex then a new vertex v with *associated_point(v)* = x is added to DT. In either case, v is returned.

Simplex_handle *DT.locate(Point_d x)*

returns a simplex of the nearest site triangulation containing x in its closure (returns *Simplex_handle()* if x lies outside the convex hull of S).

Vertex_handle *DT.lookup(Point_d x)*

if DT contains a vertex v with *associated_point(v)* = x the result is v otherwise the result is *Vertex_handle()*.

Vertex_handle *DT.nearest_neighbor(Point_d x)*

computes a vertex v of DT that is closest to x, i.e., $dist(x, associated\_point(v)) = \min\{dist(x,\ associated\_point(u)) \mid u \in S\ \}$.

std::list<Vertex_handle>

DT.range_search(Sphere_d C)

returns the list of all vertices contained in the closure of sphere C.

std::list<Vertex_handle>

DT.range_search(std::vector<Point_d> A)

returns the list of all vertices contained in the closure of the simplex whose corners are given by A.
Precondition: A must consist of $d+1$ affinely independent points in base space.

std::list<Simplex_handle>

DT.all_simplices(Delaunay_voronoi_kind k = NEAREST)

returns a list of all simplices of either the nearest or the furthest site Delaunay triangulation of S.

std::list<Vertex_handle>

DT.all_vertices(Delaunay_voronoi_kind k = NEAREST)

returns a list of all vertices of either the nearest or the furthest site Delaunay triangulation of *S*.

std::list<Point_d>	*DT.all_points()*	returns *S*.
Point_const_iterator	*DT.points_begin()*	returns the start iterator for points in *DT*.
Point_const_iterator	*DT.points_end()*	returns the past the end iterator for points in *DT*.

Simplex_iterator *DT.simplices_begin(Delaunay_voronoi_kind k = NEAREST)*

returns the start iterator for simplices of *DT*.

Simplex_iterator	*DT.simplices_end()*	returns the past the end iterator for simplices of *DT*.

Implementation

The data type is derived from *Convex_hull_d* via the lifting map. For a point *x* in *d*-dimensional space let *lift(x)* be its lifting to the unit paraboloid of revolution. There is an intimate relationship between the Delaunay triangulation of a point set *S* and the convex hull of *lift(S)*: The nearest site Delaunay triangulation is the projection of the lower hull and the furthest site Delaunay triangulation is the upper hull. For implementation details we refer the reader to the implementation report available from the CGAL server.

The space requirement is the same as for convex hulls. The time requirement for an insert is the time to insert the lifted point into the convex hull of the lifted points.

Example

The abstract data type *Delaunay_d* has a default instantiation by means of the *d*-dimensional geometric kernel.

```
#include <CGAL/Homogeneous_d.h>
#include <CGAL/leda_integer.h>
#include <CGAL/Delaunay_d.h>

typedef leda_integer RT;
typedef CGAL::Homogeneous_d<RT> Kernel;
typedef CGAL::Delaunay_d<Kernel> Delaunay_d;
typedef Delaunay_d::Point_d Point;
typedef Delaunay_d::Simplex_handle Simplex_handle;
typedef Delaunay_d::Vertex_handle Vertex_handle;

int main()
{
  Delaunay_d T(2);
  Vertex_handle v1 = T.insert(Point_d(2,11));
  ...
}
```

Traits requirements

Delaunay_d< R, Lifted_R > requires the following types from the kernel traits *Lifted_R*:

```
RT Point_d Vector_d Ray_d Hyperplane_d
```

and uses the following function objects from the kernel traits:

```
Construct_hyperplane_d
Construct_vector_d
Vector_to_point_d / Point_to_vector_d
Orientation_d
Orthogonal_vector_d
Oriented_side_d / Has_on_positive_side_d
Affinely_independent_d
Contained_in_simplex_d
Contained_in_affine_hull_d
Intersect_d
Lift_to_paraboloid_d / Project_along_d_axis_d
Component_accessor_d
```

Delaunay_d< R, Lifted_R > requires the following types from the kernel traits *R*:

```
FT Point_d Sphere_d
```

and uses the following function objects from the kernel traits *R*:

```
Construct_sphere_d
Squared_distance_d
Point_of_sphere_d
Affinely_independent_d
Contained_in_simplex_d
```

Low Dimensional Output Routines

template <typename R, typename Lifted_R>
template <typename R, typename Lifted_R> void

d2_map(Delaunay_d<R,Lifted_R> D,
GRAPH< typename Delaunay_d<R,Lifted_R>::Point_d, int >& DTG,
typename Delaunay_d<R,Lifted_R>::Delaunay_voronoi_kind k = Delaunay_d<R,Lifted_R>::NEAREST)

constructs a LEDA graph representation of the nearest (*kind = NEAREST* or the furthest (*kind = FURTHEST*) site Delaunay triangulation.

Precondition: *dim() == 2*.

Part VI

Polygons

Chapter 18

2D Polygons

Geert-Jan Giezeman and Wieger Wesselink

Contents

18.1 Introduction

A polygon is a closed chain of edges. Several algorithms are available for polygons. For some of those algorithms, it is necessary that the polygon is simple. A polygon is simple if edges don't intersect, except consecutive edges, which intersect in their common vertex.

The following algorithms are available:

- find the leftmost, rightmost, topmost and bottommost vertex.
- compute the (signed) area.
- check if a polygon is simple.
- check if a polygon is convex.
- find the orientation (clockwise or counterclockwise)
- check if a point lies inside a polygon.

All those operations take two forward iterators as parameters in order to describe the polygon. These parameters have a point type as value type.

The type *Polygon_2* can be used to represent polygons. Polygons are dynamic. Vertices can be modified, inserted and erased. They provide the algorithms described above as member functions. Moreover, they provide ways of iterating over the vertices and edges.

Currently, the *Polygon_2* class is a nice wrapper around a container of points, but little more. Especially, computed values are not cached. That is, when the *is_simple()* member function is called twice or more, the result is computed each time anew.

18.2 Examples

18.2.1 The Polygon Class

The following example creates a polygon and illustrates the usage of some member functions.

```
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/Polygon_2.h>
#include <iostream>

typedef CGAL::Exact_predicates_inexact_constructions_kernel K;
typedef K::Point_2 Point;
typedef CGAL::Polygon_2<K> Polygon_2;
using std::cout; using std::endl;


int main()
{
  Point points[] = { Point(0,0), Point(5.1,0), Point(1,1), Point(0.5,6)};
  Polygon_2 pgn(points, points+4);

  // check if the polygon is simple.
  cout << "The polygon is " <<
    (pgn.is_simple() ? "" : "not ") << "simple." << endl;

  // check if the polygon is convex
  cout << "The polygon is " <<
    (pgn.is_convex() ? "" : "not ") << "convex." << endl;

  return 0;
}
```

File: `examples/Polygon/Polygon.cpp`

18.2.2 Algorithms Operating on Sequences of Points

The following example creates a polygon and illustrates the usage of some global functions that operate on sequences of points.

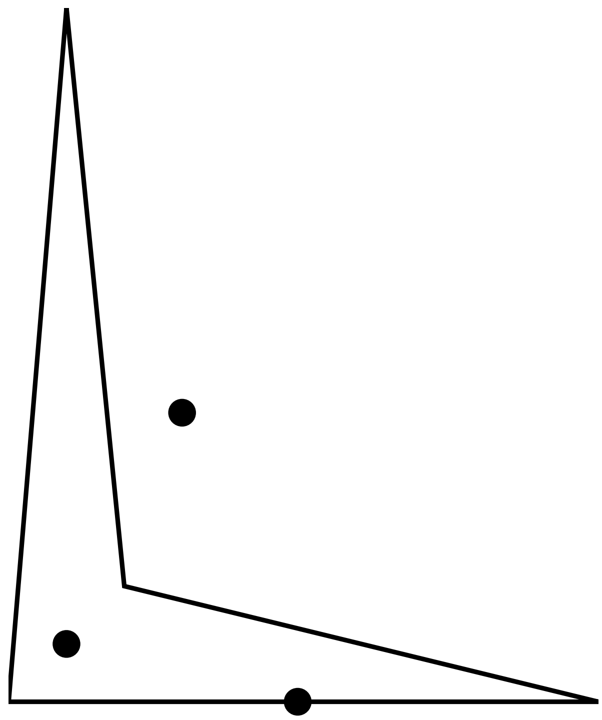


Figure 18.1: A polygon and some points

```
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/Polygon_2_algorithms.h>
#include <iostream>

typedef CGAL::Exact_predicates_inexact_constructions_kernel K;
typedef K::Point_2 Point;
using std::cout; using std::endl;

void check_inside(Point pt, Point *pgn_begin, Point *pgn_end, K traits)
{
  cout << "The point " << pt;
  switch(CGAL::bounded_side_2(pgn_begin, pgn_end,pt, traits)) {
    case CGAL::ON_BOUNDED_SIDE :
      cout << " is inside the polygon.\n";
      break;
    case CGAL::ON_BOUNDARY:
      cout << " is on the polygon boundary.\n";
      break;
    case CGAL::ON_UNBOUNDED_SIDE:
      cout << " is outside the polygon.\n";
      break;
  }
}

int main()
{
  Point points[] = { Point(0,0), Point(5.1,0), Point(1,1), Point(0.5,6)};
```

```
  // check if the polygon is simple.
  cout << "The polygon is "
    << (CGAL::is_simple_2(points, points+4, K()) ? "" : "not ")
    << "simple." << endl;

  check_inside(Point(0.5, 0.5), points, points+4, K());
  check_inside(Point(1.5, 2.5), points, points+4, K());
  check_inside(Point(2.5, 0), points, points+4, K());

  return 0;
}
```

File: examples/Polygon/polygon_algorithms.cpp

18.2.3 Polygons in 3D Space

Sometimes it is useful to run a 2D algorithm on 3D data. Polygons may be contours of a 3D object, where the contours are organized in parallel slices, generated by segmentation of image data from a scanner.

In order to avoid an explixit projection on the *xy* plane, one can use the traits class *Projection_traits_xy_3<K>* which is part of the 2D and 3D Linear Geometric Kernel,

```
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/Projection_traits_yz_3.h>
#include <CGAL/Polygon_2_algorithms.h>
#include <iostream>

typedef CGAL::Exact_predicates_inexact_constructions_kernel K;
typedef K::Point_3 Point_3;

int main()
{
  Point_3 points[4] = { Point_3(0,1,1), Point_3(0,2,1), Point_3(0,2,2),
   Point_3(0,1,2) };
  bool b =  CGAL::is_simple_2(points,
                              points+4,
                              CGAL::Projection_traits_yz_3<K>());
  if (!b){
    std::cerr << "Error polygon is not simple" << std::endl;
    return 1;
  }
  return 0;
}
```

File: examples/Polygon/projected_polygon.cpp

2D Polygons
Reference Manual

Geert-Jan Giezeman and Wieger Wesselink

Assertions

The assertion flags for the polygons and polygon operations use *POLYGON* in their names (*e.g.*, *CGAL_POLYGON_NO_ASSERTIONS*).

18.3 Classified Reference Pages

Concepts

Classes

Global Functions

18.4 Alphabetical List of Reference Pages

CGAL::area_2

Function

Definition

The function *area_2* computes the signed area of a polygon.

#include <CGAL/Polygon_2_algorithms.h>

```
template <class ForwardIterator, class Traits>
void                area_2( ForwardIterator first,
                            ForwardIterator last,
                            typename Traits::FT &result,
                            Traits traits)
```

Computes the signed area of the polygon defined by the range of points *first* ... *last*. The area is returned in the parameter *result*. The sign is positive for counterclockwise polygons, negative for clockwise polygons. If the polygon is not simple, the area is not well defined. The functionality is also available by the *polygon_area_2* function, which returns the area instead of taking it as a parameter.

Requirements

1. *Traits* is a model of the concept *PolygonTraits_2*. Only the following members of this traits class are used:
   - *Compute_area_2* : Computes the signed area of the oriented triangle defined by 3 *Point_2* passed as arguments.
   - *FT*
   - *compute_area_2_object()*
2. The value type of *ForwardIterator* must be *Traits::Point_2*,

See Also

CGAL::polygon_area_2 . . . . page 1479
PolygonTraits_2 . . . . page 1470
CGAL::orientation_2 . . . . page 1468
CGAL::Polygon_2<PolygonTraits_2, Container> . . . . page 1472

Function

CGAL::bbox_2

Definition

#include <CGAL/Polygon_2_algorithms.h>

template <class InputIterator, class Traits>
Bbox_2 *bbox_2(InputIterator first, InputIterator last, Traits traits)*

Returns the bounding box of the range *[first,last)*.

Requirements

1. *Traits* is a model of the concept *PolygonTraits_2*. In fact, only the members *Construct_bbox_2* and *construct_bbox_2_object()* are used.
2. The value type of *InputIterator* must be *Traits::Point_2*,

See Also

CGAL::Polygon_2<PolygonTraits_2, Container> . . . . page 1472

CGAL::bottom_vertex_2

Function

Definition

#include <CGAL/Polygon_2_algorithms.h>

template <class ForwardIterator, class Traits>
ForwardIterator *bottom_vertex_2(ForwardIterator first, ForwardIterator last, Traits traits)*

Returns an iterator to the bottommost point from the range *[first,last)*. In case of a tie, the point with the smallest x-coordinate is taken.

Requirements

1. *Traits* is a model of the concept *PolygonTraits_2*. In fact, only the members *Less_yx_2* and *less_yx_2_object()* are used.
2. The value type of *ForwardIterator* must be *Traits::Point_2*,

See Also

CGAL::left_vertex_2 . . . . page 1467
CGAL::right_vertex_2 . . . . page 1480
CGAL::top_vertex_2 . . . . page 1481
CGAL::Polygon_2<PolygonTraits_2, Container> . . . . page 1472
PolygonTraits_2 . . . . page 1470

Example

CGAL::bounded_side_2

Definition

The function *bounded_side_2* computes if a point lies inside a polygon.

#include <CGAL/Polygon_2_algorithms.h>

```
template <class ForwardIterator, class Traits>
Bounded_side        bounded_side_2( ForwardIterator first,
                                    ForwardIterator last,
                                    Traits::Point_2 point,
                                    Traits traits)
```

The function *bounded_side_2* computes if a point lies inside a polygon. The polygon is defined by the sequence of points *first*...*last*. Being inside is defined by the odd-even rule. If we take a ray starting at the point and extending to infinity (in any direction), we count the number of intersections. If this number is odd, the point is inside, otherwise it is outside. If the point is on a polygon edge, a special value is returned. A simple polygon divides the plane in an unbounded and a bounded region. According to the definition points in the bounded region are inside the polygon.

Requirements

1. *Traits* is a model of the concept *PolygonTraits_2*. Only the following members of this traits class are used:
   - *Compare_x_2*
   - *Compare_y_2*
   - *Orientation_2*
   - *compare_x_2_object()*
   - *compare_y_2_object()*
   - *orientation_2_object()*
2. The value type of *ForwardIterator* must be *Traits::Point_2*,

Implementation

The running time is linear in the number of vertices of the polygon. A horizontal ray is taken to count the number of intersections. Special care is taken that the result is correct even if there are degeneracies (if the ray passes through a vertex).

See Also

PolygonTraits_2 .... page 1470
CGAL::oriented_side_2 .... page 1469
CGAL::Polygon_2<PolygonTraits_2, Container> .... page 1472
CGAL::Bounded_side

CGAL::is_convex_2

Definition

The function *is_convex_2* computes if a polygon is convex.

#include <CGAL/Polygon_2_algorithms.h>

template <class ForwardIterator, class Traits>
bool *is_convex_2(ForwardIterator first, ForwardIterator last, Traits traits)*

Requirements

1. *Traits* is a model of the concept *PolygonTraits_2*. Only the following members of this traits class are used:
   - *Less_xy_2*
   - *Orientation_2*
   - *less_xy_2_object*
   - *orientation_2_object*
2. *ForwardIterator::value_type* should be *Traits::Point_2*,

See Also

PolygonTraits_2 . . . . page 1470
CGAL::Polygon_2<PolygonTraits_2, Container> . . . . page 1472

Function

CGAL::is_simple_2

Definition

The function *is_simple_2* computes if a polygon is simple, that is, if the edges do not intersect, except consecutive edges in their common vertex.

#include <CGAL/Polygon_2_algorithms.h>

template <class ForwardIterator, class Traits>
inline bool *is_simple_2(ForwardIterator first, ForwardIterator last, Traits traits)*

Checks if the polygon defined by the iterator range *first* ... *last* is simple.

Requirements

1. *Traits* is a model of the concept *PolygonTraits_2*. Only the following members of this traits class are used:
   - *Point_2*
   - *Less_xy_2*
   - *Orientation_2*
   - *less_xy_2_object()*
   - *orientation_2_object()*
2. The value type of *ForwardIterator* must be *Traits::Point_2*,

Implementation

The simplicity test is implemented by means of a plane sweep algorithm. The algorithm is quite robust when used with inexact number types. The running time is O(n log n), where n is the number of vertices of the polygon.

See Also

PolygonTraits_2 .... page 1470
CGAL::Polygon_2<PolygonTraits_2, Container> .... page 1472

CGAL::left_vertex_2

Function

Definition

#include <CGAL/Polygon_2_algorithms.h>

template <class ForwardIterator, class Traits>
ForwardIterator *left_vertex_2(ForwardIterator first, ForwardIterator last, Traits traits)*

Returns an iterator to the leftmost point from the range *[first,last)*. In case of a tie, the point with the smallest y-coordinate is taken.

Requirements

1. *Traits* is a model of the concept *PolygonTraits_2*. In fact, only the members *Less_xy_2* and *less_xy_2_object()* are used.
2. The value type of *ForwardIterator* must be *Traits::Point_2*,

See Also

CGAL::right_vertex_2 . . . page 1480
CGAL::top_vertex_2 . . . page 1481
CGAL::bottom_vertex_2 . . . page 1463
CGAL::Polygon_2<PolygonTraits_2, Container> . . . page 1472

Function

CGAL::orientation_2

Definition

The function *orientation_2* computes if a polygon is clockwise or counterclockwise oriented.

#include <CGAL/Polygon_2_algorithms.h>

template <class ForwardIterator, class Traits>
Orientation *orientation_2(ForwardIterator first, ForwardIterator last, Traits traits)*

Precondition: *is_simple_2(first, last, traits);*

Requirements

1. *Traits* is a model of the concept *PolygonTraits_2*. Only the following members of this traits class are used:
   - *Less_xy_2*
   - *less_xy_2_object()*
   - *orientation_2_object()*
2. The value type of *ForwardIterator* must be *Traits::Point_2*,

See Also

PolygonTraits_2 . . . . page 1470
CGAL::is_simple_2 . . . . page 1466
CGAL::Polygon_2<PolygonTraits_2, Container> . . . . page 1472
CGAL::Orientation

CGAL::oriented_side_2

Definition

The function *oriented_side_2* computes on which side of a polygon a point lies.

#include <CGAL/Polygon_2_algorithms.h>

```
template <class ForwardIterator, class Traits>
Oriented_side        oriented_side_2( ForwardIterator first,
                                      ForwardIterator last,
                                      Traits::Point_2 point,
                                      Traits traits)
```

Requirements

1. *Traits* is a model of the concept *PolygonTraits_2*. Only the following members of this traits class are used:
   - *Less_xy_2*
   - *Compare_x_2*
   - *Compare_y_2*
   - *Orientation_2*
   - *less_xy_2_object()*
   - *compare_x_2_object()*
   - *compare_y_2_object()*
   - *orientation_2_object()*
2. The value type of *ForwardIterator* must be *Traits::Point_2*,

See Also

PolygonTraits_2 . . . . page 1470
CGAL::bounded_side_2 . . . . page 1464
CGAL::is_simple_2 . . . . page 1466
CGAL::Polygon_2<PolygonTraits_2, Container> . . . . page 1472
CGAL::Oriented_side

Concept

PolygonTraits_2

Definition

The *Polygon_2* class and the functions that implement the functionality found in that class each are parameterized by a traits class that defines the primitives used in the algorithms. The concept PolygonTraits_2 defines this common set of requirements.

The requirements of PolygonTraits_2 are a subset of the kernel requirements. We only list the types and methods which are required and refer to the description of the kernel concept for details.

Types

PolygonTraits_2:: FT	
PolygonTraits_2:: Point_2	The point type.
PolygonTraits_2:: Segment_2	The segment type.
PolygonTraits_2:: Construct_segment_2	
PolygonTraits_2:: Equal_2	
PolygonTraits_2:: Less_xy_2	
PolygonTraits_2:: Less_yx_2	
PolygonTraits_2:: Compare_x_2	
PolygonTraits_2:: Compare_y_2	
PolygonTraits_2:: Orientation_2	
PolygonTraits_2:: Compute_area_2	Computes the signed area of the oriented triangle defined by 3 *Point_2* passed as arguments.

Creation

A default constructor and copy constructor are required.

Operations

The following functions that create instances of the above predicate object types must exist.

Equal_2	*traits.equal_2_object()*

Less_xy_2	*traits.less_xy_2_object()*
Less_yx_2	*traits.less_yx_2_object()*
Compare_y_2	*traits.compare_y_2_object()*
Compare_x_2	*traits.compare_x_2_object()*
Orientation_2	*traits.orientation_2_object()*
Compute_area_2	*traits.compute_area_2_object()*
Construct_segment_2	*traits.construct_segment_2_object()*

Has Models

The kernels supplied by CGAL are models of PolygonTraits_2.
CGAL::Projection_traits_xy_3<K> . . . . page 604
CGAL::Projection_traits_yz_3<K>
CGAL::Projection_traits_zx_3<K>

See Also

CGAL::Polygon_2<PolygonTraits_2, Container> . . . . page 1472

Class

CGAL::Polygon_2<PolygonTraits_2, Container>

Definition

The class *Polygon_2<PolygonTraits_2, Container>* implements polygons. The *Polygon_2<PolygonTraits_2, Container>* is parameterized by a traits class and a container class. The latter can be any class that fulfills the requirements for an STL container. It defaults to the vector class.

#include <CGAL/Polygon_2.h>

Types

Polygon_2<PolygonTraits_2, Container>:: Traits The traits type.

Polygon_2<PolygonTraits_2, Container>:: Container

The container type.

typedef Traits::FT	*FT;*	The number type, which is the *field type* of the points of the polygon.
typedef Traits::Point_2	*Point_2;*	The point type of the polygon.
typedef Traits::Segment_2	*Segment_2;*	The type of a segment between two points of the polygon.

The following types denote iterators that allow to traverse the vertices and edges of a polygon. Since it is questionable whether a polygon should be viewed as a circular or as a linear data structure both circulators and iterators are defined. The circulators and iterators are non-mutable.[1] The iterator category is in all cases bidirectional, except for *Vertex_iterator*, which has the same iterator category as *Container::iterator*. **N.B.** In fact all of them should have the same iterator category as *Container::iterator*. However, due to compiler problems this is currently not possible.

For vertices we define

Polygon_2<PolygonTraits_2, Container>:: Vertex_iterator
Polygon_2<PolygonTraits_2, Container>:: Vertex_circulator

Their value type is *Point_2*.

For edges we define

Polygon_2<PolygonTraits_2, Container>:: Edge_const_circulator
Polygon_2<PolygonTraits_2, Container>:: Edge_const_iterator

Their value type is *Segment_2*.

[1] At least conceptually. The enforcement depends on preprocessor flags.

Creation

Polygon_2<PolygonTraits_2, Container> pgn(Traits p_traits = Traits());

Creates an empty polygon *pgn*.

template <class InputIterator>
Polygon_2<PolygonTraits_2, Container> pgn(InputIterator first, InputIterator last, Traits p_traits = Traits())

Introduces a polygon *pgn* with vertices from the sequence defined by the range *[first,last)*. The value type of *InputIterator* must be *Point_2*.

Modifiers

void *pgn.set(Vertex_iterator pos, Point_2 x)*

Acts as **pos* = *x*, except that that would be illegal because the iterator is not mutable.

Vertex_iterator *pgn.insert(Vertex_iterator i, Point_2 q)*

Inserts the vertex *q* before *i*. The return value points to the inserted vertex.

template <class InputIterator>
void *pgn.insert(Vertex_iterator i, InputIterator first, InputIterator last)*

Inserts the vertices in the range *[first, last)* before *i*. The value type of points in the range *[first,last)* must be *Point_2*.

void *pgn.push_back(Point_2 q)*

Has the same semantics as *p.insert(p.vertices_end(), q)*.

void *pgn.erase(Vertex_iterator i)*

Erases the vertex pointed to by *i*.

void *pgn.erase(Vertex_iterator first, Vertex_iterator last)*

Erases the vertices in the range *[first, last)*.

void *pgn.clear()* Erases all vertices.

void *pgn.reverse_orientation()*

Reverses the orientation of the polygon. The vertex pointed to by *p.vertices_begin()* remains the same.

Access Functions

The following methods of the class return circulators and iterators that allow to traverse the vertices and edges.

Vertex_iterator *pgn.vertices_begin() const*

Returns a constant iterator that allows to traverse the vertices of the polygon *pgn*.

Vertex_iterator *pgn.vertices_end() const*

Returns the corresponding past-the-end iterator.

Vertex_circulator *pgn.vertices_circulator() const*

Returns a mutable circulator that allows to traverse the vertices of the polygon *pgn*.

Edge_const_iterator *pgn.edges_begin() const*

Returns a non-mutable iterator that allows to traverse the edges of the polygon *pgn*.

Edge_const_iterator	*pgn.edges_end() const*	
		Returns the corresponding past-the-end iterator.
Edge_const_circulator	*pgn.edges_circulator() const*	
		Returns a non-mutable circulator that allows to traverse the edges of the polygon *pgn*.

Predicates

bool	*pgn.is_simple() const*	Returns whether *pgn* is a simple polygon.
bool	*pgn.is_convex() const*	Returns whether *pgn* is convex.
Orientation	*pgn.orientation() const*	
		Returns the orientation of *pgn*. If the number of vertices $p.size() < 3$ then *COLLINEAR* is returned. *Precondition*: *p.is_simple()*.
Oriented_side	*pgn.oriented_side(Point_2 q) const*	
		Returns *POSITIVE_SIDE*, or *NEGATIVE_SIDE*, or *ON_ORIENTED_BOUNDARY*, depending on where point *q* is. *Precondition*: *p.is_simple()*.
Bounded_side	*pgn.bounded_side(Point_2 q) const*	
		Returns the symbolic constant *ON_BOUNDED_SIDE*, *ON_BOUNDARY* or *ON_UNBOUNDED_SIDE*, depending on where point *q* is. *Precondition*: *p.is_simple()*.
Bbox_2	*pgn.bbox() const*	Returns the smallest bounding box containing *pgn*.
Traits::FT	*pgn.area() const*	Returns the signed area of the polygon *pgn*. This means that the area is positive for counter clockwise polygons and negative for clockwise polygons.
Vertex_iterator	*pgn.left_vertex()*	Returns the leftmost vertex of the polygon *pgn* with the smallest *x*-coordinate.
Vertex_iterator	*pgn.right_vertex()*	Returns the rightmost vertex of the polygon *pgn* with the largest *x*-coordinate.

Vertex_iterator	*pgn.top_vertex()*	Returns topmost vertex of the polygon *pgn* with the largest *y*-coordinate.
Vertex_iterator	*pgn.bottom_vertex()*	Returns the bottommost vertex of the polygon *pgn* with the smallest *y*-coordinate.

For convenience we provide the following Boolean functions:

bool *pgn.is_counterclockwise_oriented() const*
bool *pgn.is_clockwise_oriented() const*

bool *pgn.is_collinear_oriented() const*
bool *pgn.has_on_positive_side(Point_2 q) const*
bool *pgn.has_on_negative_side(Point_2 q) const*
bool *pgn.has_on_boundary(Point_2 q) const*
bool *pgn.has_on_bounded_side(Point_2 q) const*
bool *pgn.has_on_unbounded_side(Point_2 q) const*

Random access methods

These methods are only available for random access containers.

Point_2 *pgn.vertex(std::size_t i) const*

Returns a (const) reference to the *i*-th vertex.

Point_2 *pgn*[*std::size_t i*] Returns a (const) reference to the *i*-th vertex.

Segment_2 *pgn.edge(std::size_t i) const*

Returns the *i*-th edge.

Miscellaneous

std::size_t	*pgn.size() const*	Returns the number of vertices of the polygon *pgn*.
bool	*pgn.is_empty() const*	Returns $p.size() == 0$.
Container	*pgn.container() const*	Returns a const reference to the sequence of vertices of the polygon *pgn*.

Globally defined operators

template <class Traits, class Container1, class Container2>

bool *Polygon_2<Traits,Container1> p1 == Polygon_2<Traits,Container2> p2*

Test for equality: two polygons are equal iff there exists a cyclic permutation of the vertices of *p2* such that they are equal to the vertices of *p1*. Note that the template argument *Container* of *p1* and *p2* may be different.

template <class Traits, class Container1, class Container2>

bool *Polygon_2<Traits,Container1> p1* $!=$ *Polygon_2<Traits,Container2> p2*

Test for inequality.

template <class Transformation, class Traits, class Container>

Polygon_2<Traits,Container>

transform(Transformation t, Polygon_2<Traits,Container> p)

Returns the image of the polygon p under the transformation t.

I/O

The I/O operators are defined for *iostream*. The format for the iostream is an internal format.

ostream& *ostream& os* $<<$ *Polygon_2<Traits,Container> p*

Inserts the polygon *pgn* into the stream *os*.
Precondition: The insert operator must be defined for *Point_2*.

istream& *istream& is* $>>$ *Polygon_2<Traits,Container> p*

Reads a polygon from stream *is* and assigns it to *pgn*.
Precondition: The extract operator must be defined for *Point_2*.

The information output in the *iostream* is the number of points followed by the output of the coordinates of the vertices.

Implementation

The methods *is_simple*, *is_convex*, *orientation*, *oriented_side*, *bounded_side*, *bbox*, *area*, *left_vertex*, *right_vertex*, *top_vertex* and *bottom_vertex* are all implemented using the algorithms on sequences of 2D points. See the corresponding global functions for information about which algorithms were used and what complexity they have.

Example

The following code fragment creates a polygon and checks if it is convex.

```
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/Polygon_2.h>
#include <iostream>

typedef CGAL::Exact_predicates_inexact_constructions_kernel K;
typedef K::Point_2 Point;
typedef CGAL::Polygon_2<K> Polygon_2;
```

```
using std::cout; using std::endl;


int main()
{
  Point points[] = { Point(0,0), Point(5.1,0), Point(1,1), Point(0.5,6)};
  Polygon_2 pgn(points, points+4);

  // check if the polygon is simple.
  cout << "The polygon is " <<
    (pgn.is_simple() ? "" : "not ") << "simple." << endl;

  // check if the polygon is convex
  cout << "The polygon is " <<
    (pgn.is_convex() ? "" : "not ") << "convex." << endl;

  return 0;
}
```

File: examples/Polygon/Polygon.cpp

CGAL::polygon_area_2

Function

Definition

The function *polygon_area_2* computes the signed area of a polygon.

#include <CGAL/Polygon_2_algorithms.h>

template <class ForwardIterator, class Traits>
typename Traits::FT *polygon_area_2(ForwardIterator first, ForwardIterator last, Traits traits)*

Computes the signed area of the polygon defined by the range of points *first* ... *last*. The sign is positive for counterclockwise polygons, negative for clockwise polygons. If the polygon is not simple, the area is not well defined.

Requirements

1. *Traits* is a model of the concept *PolygonTraits_2*. Only the following members of this traits class are used:
   - *Compute_area_2* : Computes the signed area of the oriented triangle defined by 3 *Point_2* passed as arguments.
   - *FT*
   - *compute_area_2_object*
2. *ForwardIterator::value_type* should be *Traits::Point_2*,

See Also

PolygonTraits_2 .... page 1470
CGAL::orientation_2 .... page 1468
CGAL::Polygon_2<PolygonTraits_2, Container> .... page 1472

Function

CGAL::right_vertex_2

Definition

#include <CGAL/Polygon_2_algorithms.h>

template <class ForwardIterator, class Traits>
ForwardIterator *right_vertex_2(ForwardIterator first, ForwardIterator last, Traits traits)*

Returns an iterator to the rightmost point from the range *[first,last)*. In case of a tie, the point with the largest y-coordinate is taken.

Requirements

1. *Traits* is a model of the concept *PolygonTraits_2*. In fact, only the members *Less_xy_2* and *less_xy_2_object()* are used.
2. The value type of *ForwardIterator* must be *Traits::Point_2*,

See Also

CGAL::left_vertex_2 . . . . page 1467
CGAL::top_vertex_2 . . . . page 1481
CGAL::bottom_vertex_2 . . . . page 1463
CGAL::Polygon_2<PolygonTraits_2, Container> . . . . page 1472

CGAL::top_vertex_2

Definition

#include <CGAL/Polygon_2_algorithms.h>

template <class ForwardIterator, class Traits>
ForwardIterator *top_vertex_2(ForwardIterator first, ForwardIterator last, Traits traits)*

Returns an iterator to the topmost point from the range *[first,last)*. In case of a tie, the point with the largest x-coordinate is taken.

Requirements

1. *Traits* is a model of the concept *PolygonTraits_2*. In fact, only the members *Less_yx_2* and *less_yx_2_object()* are used.
2. The value type of *ForwardIterator* must be *Traits::Point_2*,

See Also

CGAL::left_vertex_2 . . . . page 1467
CGAL::right_vertex_2 . . . . page 1480
CGAL::bottom_vertex_2 . . . . page 1463
CGAL::Polygon_2<PolygonTraits_2, Container> . . . . page 1472

Chapter 19

2D Regularized Boolean Set-Operations

Efi Fogel, Ophir Setter, Ron Wein, Guy Zucker, Baruch Zukerman, and Dan Halperin

Contents

19.1 Introduction

This package consists of the implementation of Boolean set-operations on point sets bounded by weakly x-monotone curves[1] in 2-dimensional Euclidean space. In particular, it contains the implementation of *regularized* Boolean set-operations, intersection predicates, and point containment predicates. Figure 35 shows simple examples of such operations.

Ordinary Boolean set-operations, which distinguish between the interior and the boundary of a polygon, are not implemented within this package. The *Nef_2* package supports these operations for (linear) polygons; see Chapter 20.

[1] A continuous planar curve C is *weakly x-monotone* if every vertical line intersects it at most once, or if it is vertical. Hereafter we refer to weakly x-monotone curves as x-monotone curves.

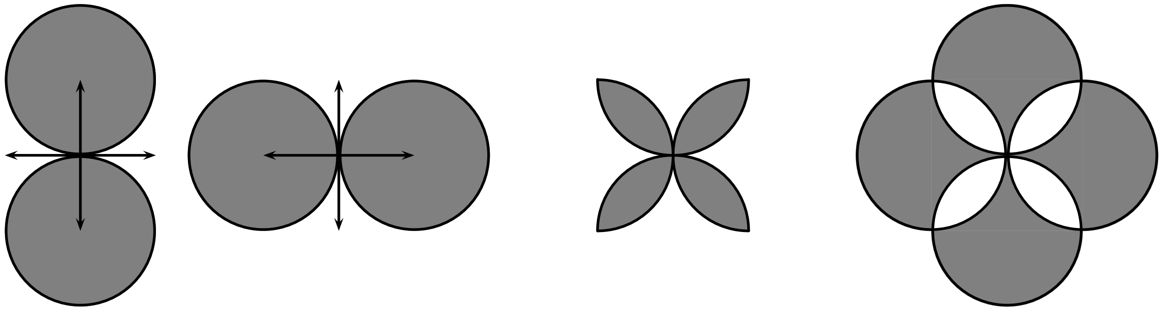


Figure 19.1: Examples of Boolean set-operations on general polygons.

In the rest of this chapter we use, unless otherwise stated, the traditional notation to denote regularized operations; e.g., $P \cap Q$ means the *regularized* intersection of P and Q.

Our package supports the following Boolean set-operations on two point sets P and Q that each is the union of one or more general polygons:

Intersection computes the intersection $R = P \cap Q$.

Join computes the union $R = P \cup Q$.

Difference computes the difference $R = P \setminus Q$.

Symmetric Difference computes the symmetric difference $P = P \oplus Q = (P \setminus Q) \cup (Q \setminus P)$.

Complement computes the complement $R = \overline{P}$.

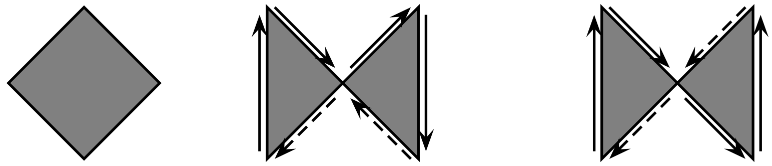
Intersection predicate tests whether the two sets P and Q overlap, distinguishing three possible scenarios: (i) the two sets intersect on their interior (that is, their regularized intersection is not empty $P \cap Q \neq \emptyset$); (ii) the boundaries of two sets intersect but their interiors are disjoint; namely they have a finite number of common points or even share a boundary curve (still in this case $P \cap Q = \emptyset$; and (iii) the two sets are disjoint.

In general, the set R, resulting from a regularized Boolean set-operation, is considered as being a closed point-set; see definition of regularized boolean set operations below.

In the rest of this chapter we review the Boolean set-operations package in more depth. In Section 19.3 we focus on Boolean set-operations on linear polygons, introducing the notion of polygons with holes and of a general polygon set. Section 19.4 introduces general polygons. We first discuss polygons whose edges are either line segments or circular arcs and then explain how to construct and use general polygons whose edges can be arbitrary weakly x-monotone curves.

19.2 Terms and Definitions

- The counterclockwise cyclic sequence of alternating polygon edges and polygon vertices is referred to as the polygon **(outer) boundary**.



A *B* *C*

Figure 19.2: Examples of polygons. (a) A simple polygon. (b) A relatively simple polygon (c) A polygon that is neither simple nor relatively simple

- A polygon whose curves are pairwise disjoint in their interior, and whose vertices' degree equals two is defined as a **Simple polygon**. Such a polygon has a well-defined interior and exterior and is topologically equivalent to a disk. Note that while traversing the edges of the relatively simple polygon illustrated above (B), no curve is crossed over.
- A **Relatively simple** polygon allows vertices with a degree>2, but all of its edges are disjoint in their interior. Furthermore, it must be an orientable polygon. Namely when it is inserted into an arrangement and its outer boundary is traversed, the same face is adjacent to all of the halfedges (no crossing over any curve during the traversal). Note that while polygon C has the same curves as polygon B, traversal of the curves leads to crossing over a previously traversed curve, and is therefore neither simple nor relatively simple.
- A polygon in our context must be relatively simple and its outer boundary vertices must be ordered in a counterclockwise direction around the interior of the polygon. We extend the notion of a polygon to a **point set** in IR^2 that has a topology of a polygon and its boundary edges must map to x-monotone curves, and refer to it as a **general polygon**. We sometimes use the term *polygon* instead of general polygon for simplicity hereafter.
- A **polygon with holes** represents a point set that may be bounded or unbounded. In case of a bounded set, its *outer boundary* is represented as a *relatively simple* (but not necessarily simple) polygon, whose vertices are oriented in a *counterclockwise* order around the interior of the set. In addition, the set may contain *holes*, where each hole is represented as a *simple* polygon, whose vertices are oriented in a *clockwise* order around the interior of the hole. Note that an unbounded polygon without holes spans the entire plane. Vertices of holes may coincide with vertices of the boundary.
- A **regularized Boolean set-operation** op* can be obtained by first taking the interior of the resultant point set of an *ordinary* Boolean set-operation (P op Q) and then by taking the closure [Hof04]. That is, P op* Q = closure(interior(P op Q)). Regularized Boolean set-operations appear in Constructive Solid Geometry (CSG), because regular sets are closed under regularized Boolean set-operations, and because regularization eliminates lower dimensional features, namely isolated vertices and antennas, thus simplifying and restricting the representation to physically meaningful solids. Our package provides regularized operations on polygons and general polygons, where the edges of a general polygon may be general x-monotone curves, rather than being simple line segments.

19.2.1 Conditions for Valid Polygons

In our context, a polygon must uphold the following conditions:

1. *Closed Boundary* - the polygon's outer boundary must be a connected sequence of curves, that start and end at the same vertex.

2. *Simplicity* - the polygon must be simple.
3. *Orientation* - the polygon's outer boundary must be *counter-clockwise oriented*.

19.2.2 Conditions for Valid Polygons with Holes

In our context, a polygon with holes must uphold the following conditions:

1. *Closed Boundary* - both the outer boundary and the holes must be closed polygons as defined above.
2. *Simplicity* - the outer boundary must be a ***relatively*** *simple* polygon (as defined above). Every hole must be a *simple* polygon.
3. *Orientation* - The polygon with holes must have an outer boundary with *counter clockwise orientation*. Every hole's outer boundary should have *clockwise orientation*.
4. *The holes and the outer boundary must be pairwise disjoint,except maybe on vertices*.
   - *All holes are contained in the outer boundary* - holes must be contained inside the outer boundary and must be disjoint from it (except on vertices).
   - *Pairwise disjoint holes (on interior)* - the polygon's holes must be *pairwise disjoint*, except for intersection on a joint vertex/vertices.

19.3 Boolean Set-Operations on Linear Polygons

The basic library of CGAL includes the *Polygon_2<Kernel,Container>* class-template that represents a linear polygon in the plane. The polygon is represented by its vertices stored in a container of objects of type *Kernel::Point_2*. The polygon edges are line segments (*Kernel::Segment_2* objects) between adjacent points in the container. By default, the *Container* is a vector of *Kernel::Point_2* objects.

The following function demonstrates how to use the basic access functions of the *Polygon_2* class. It accepts a polygon *P* and prints it in a readable format:

```
template<class Kernel, class Container>
void print_polygon (const CGAL::Polygon_2<Kernel, Container>& P)
{
  typename CGAL::Polygon_2<Kernel, Container>::Vertex_const_iterator  vit;

  std::cout << "[ " << P.size() << " vertices:";
  for (vit = P.vertices_begin(); vit != P.vertices_end(); ++vit)
    std::cout << " (" << *vit << ')';
  std::cout << " ]" << std::endl;
}
```

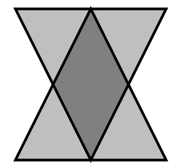
In this section we use the term *polygon* to indicate a *Polygon_2* instance, namely, a polygon having linear edges. General polygons are only discussed in Section 19.4.

The basic components of our package are the free (global) functions *complement()* that accepts a single *Polygon_2* object, and *intersection()*, *join()*,[2] *difference()*, *symmetric_difference()* and the predicate *do_intersect()* that

[2]The function that computes the union of two polygons is called *join()*, since the word *union* is reserved in C++.

accept two *Polygon_2* objects as their input. We explain how these functions should be used through several examples in the following sections.

A Simple Example



Testing whether two polygons intersect results with a Boolean value, and does not require any additional data beyond the provision of the two input polygons. The example listed below tests whether the two triangles depicted on the right intersect. It uses, as do the other example programs in this chapter, the auxiliary header file *bso_rational_nt.h*, which defines the type *Number_type* as GMP's rational number-type (*CGAL::Gmpq*), or as the number type *Quotient*<*MP_Float*> that is included in the support library of CGAL, based on whether the GMP library is installed or not. It also uses the function *print_polygon.h* listed above, which is located in the header file *print_utils.h*.

```
/*! \file do_intersect.cpp
 * Determining whether two triangles intersect.
 */

#include <CGAL/Exact_predicates_exact_constructions_kernel.h>
#include <CGAL/Boolean_set_operations_2.h>

typedef CGAL::Exact_predicates_exact_constructions_kernel Kernel;
typedef Kernel::Point_2                                   Point_2;
typedef CGAL::Polygon_2<Kernel>                           Polygon_2;

#include "print_utils.h"

int main ()
{
  Polygon_2 P;
  P.push_back (Point_2 (-1,1));
  P.push_back (Point_2 (0,-1));
  P.push_back (Point_2 (1,1));
  std::cout << "P = "; print_polygon (P);

  Polygon_2 Q;
  Q.push_back(Point_2 (-1,-1));
  Q.push_back(Point_2 (1,-1));
  Q.push_back(Point_2 (0,1));
  std::cout << "Q = "; print_polygon (Q);

  if ((CGAL::do_intersect (P, Q)))
    std::cout << "The two polygons intersect in their interior." <<
   std::endl;
  else
    std::cout << "The two polygons do not intersect." << std::endl;

  return 0;
}
```

File: examples/Boolean_set_operations_2/do_intersect.cpp

19.3.1 Polygons with Holes

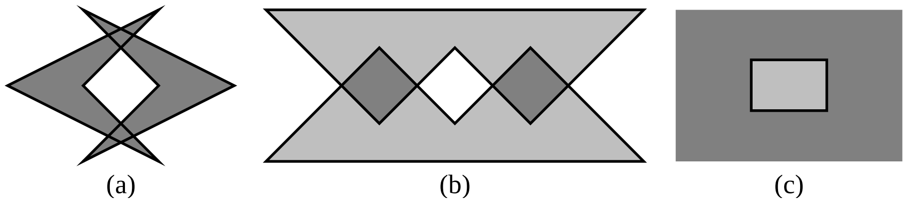


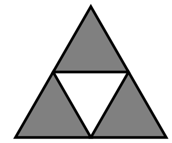
Figure 19.3: Operations on simple polygons. (a) The union of two polygons, resulting in a point set whose outer boundary is defined by a simple polygon and contains a polygonal hole in its interior. (b) The intersection (darkly shaded) of two polygons (lightly shaded), resulting in two disjoint polygons. (c) The complement (darkly shaded) of a simple polygon (lightly shaded).

In many cases a binary operation that operates on two simple polygons that have no holes may result in a set of polygons that contain holes in their interior (see Figure 19.3.1 (a)), or a set of disjoint polygons (see Figure 19.3.1 (b); a similar set may result from the union, or the symmetric difference, of two disjoint polygons). Moreover, the complement of a simple polygon is an unbounded set that contains a hole; see Figure 19.3.1 (c).

Regular sets are closed under regularized Boolean set-operations. These operations accept as input, and may result as output, polygons with holes. A *polygon with holes* represents a point set that may be bounded or unbounded. In case of a bounded set, its *outer boundary* is represented as a relatively simple (but not necessarily simple) polygon, whose vertices are oriented in a counterclockwise order around the interior of the set. In addition, the set may contain *holes*, where each hole is represented as a simple polygon, whose vertices are oriented in a clockwise order around the interior of the hole. Note that an unbounded polygon without holes spans the entire plane. Vertices of holes may coincide with vertices of the boundary; see below for an example.

A point set represented by a polygon with holes is considered to be closed. Therefore, the boundaries of the holes are parts of the set (and not part of the holes). The exact definition of the obtained polygon with holes as a result of a Boolean set-operation or a sequence of such operations is closely related to the definition of regularized Boolean set-operations, being the closure of the interior of the corresponding ordinary operation as explained next.

Consider, for example, the regular set depicted on the right, which is the result of the union of three small triangles translated appropriately. Alternatively, the same set can be reached by taking the difference between a large triangle and a small upside-down triangle. In general, there are many ways to arrive at a particular point set. However, the set of polygons with holes obtained through the application of any sequence of operations is unique. The set depicted on the right is represented as a single polygon having a triangular outer boundary with a single triangular hole in its interior — and not as three triangles that have no holes at all. As a general rule, if two point sets are connected, then they belong to the same polygon with holes.



The class template *Polygon_with_holes_2<Kernel,Container>* represents polygons with holes as described above, where the outer boundary and the hole boundaries are realized as *Polygon_2<Kernel,Container>* objects. Given an instance *P* of the *Polygon_with_holes_2* class, you can use the predicate *is_unbounded()* to check whether *P* is a unbounded set. If it is bounded, you can obtain the counterclockwise-oriented polygon that represents its outer boundary through the member function *outer_boundary()*. You can also traverse the holes of *P* through *holes_begin()* and *holes_end()*. The two functions return iterators of the nested type *Polygon_with_holes_2::Hole_const_iterator* that defines the valid range of *P*'s holes. The value type of this iterator is *Polygon_2*.

The following function demonstrates how to traverse a polygon with holes. It accepts a *Polygon_with_holes_2* object and uses the auxiliary function *print_polygon()* to prints all its components in a readable format:

```
template<class Kernel, class Container>
void print_polygon_with_holes(const CGAL::Polygon_with_holes_2<Kernel, Container> & pwh)
{
  if (! pwh.is_unbounded()) {
    std::cout << "{ Outer boundary = ";
    print_polygon (pwh.outer_boundary());
  }
  else
    std::cout << "{ Unbounded polygon." << std::endl;

  typename CGAL::Polygon_with_holes_2<Kernel,Container>::Hole_const_iterator hit;
  unsigned int k = 1;

  std::cout << "  " << pwh.number_of_holes() << " holes:" << std::endl;
  for (hit = pwh.holes_begin(); hit != pwh.holes_end(); ++hit, ++k) {
    std::cout << "    Hole #" << k << " = ";
    print_polygon (*hit);
  }
  std::cout << " }" << std::endl;
}
```

The simple versions of the free functions mentioned therefore accept two *Polygon_2* object *P* and *Q* as their input, while their output is given using polygon with holes instances:

- The complement of a simple polygon *P* is always an unbounded set with a single polygonal hole. The function *complement(P)* therefore returns a polygon-with-holes object that represents the complement of *P*.
- The union of two polygons *P* and *Q* is always a single connected set, unless of course the two input polygons are completely disjoint. In the latter case $P \cup Q$ trivially consists of the two input polygons. The free function *join(P, Q, R)* therefore returns a Boolean value indicating whether $P \cap Q \neq \emptyset$. If the two polygons are not disjoint, it assigns the polygon with holes object *R* (which it accepts by reference) with the union of the regularized union operation $P \cup Q$.

- The other three functions, namely *intersection(P, Q, oi)*, *difference(P, Q, oi)* and *symmetric_difference(P, Q, oi)*, all have a similar interface. As the result of these operation may consist of several disconnected components, they all accept an output iterator *oi*, whose value type is *Polygon_with_holes_2*, and adds the output polygons to its associated container.

Example — Joining and Intersecting Simple Polygons

The following example demonstrates the usage of the free functions *join()* and *intersect()* for computing the union and the intersection of the two simple polygons depicted in Figure 19.3.1 (b). The example uses the auxiliary function *print_polygon_with_holes()* listed above, which is located in the header file *print_utils.h* under the examples folder.

```
/*! \file simple_join_intersect.cpp
 * Computing the union and the intersection of two simple polygons.
 */

#include <CGAL/Exact_predicates_exact_constructions_kernel.h>
#include <CGAL/Boolean_set_operations_2.h>
#include <list>

typedef CGAL::Exact_predicates_exact_constructions_kernel Kernel;
typedef Kernel::Point_2                                   Point_2;
typedef CGAL::Polygon_2<Kernel>                           Polygon_2;
typedef CGAL::Polygon_with_holes_2<Kernel>
   Polygon_with_holes_2;
typedef std::list<Polygon_with_holes_2>                   Pwh_list_2;

#include "print_utils.h"

int main ()
{
  // Construct the two input polygons.
  Polygon_2 P;
  P.push_back (Point_2 (0, 0));
  P.push_back (Point_2 (5, 0));
  P.push_back (Point_2 (3.5, 1.5));
  P.push_back (Point_2 (2.5, 0.5));
  P.push_back (Point_2 (1.5, 1.5));

  std::cout << "P = "; print_polygon (P);

  Polygon_2 Q;
  Q.push_back (Point_2 (0, 2));
  Q.push_back (Point_2 (1.5, 0.5));
  Q.push_back (Point_2 (2.5, 1.5));
  Q.push_back (Point_2 (3.5, 0.5));
  Q.push_back (Point_2 (5, 2));

  std::cout << "Q = "; print_polygon (Q);

  // Compute the union of P and Q.
  Polygon_with_holes_2 unionR;
```

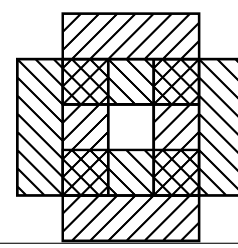
```
  if (CGAL::join (P, Q, unionR)) {
    std::cout << "The union: ";
    print_polygon_with_holes (unionR);
  } else
    std::cout << "P and Q are disjoint and their union is trivial."
              << std::endl;
  std::cout << std::endl;

  // Compute the intersection of P and Q.
  Pwh_list_2                  intR;
  Pwh_list_2::const_iterator  it;

  CGAL::intersection (P, Q, std::back_inserter(intR));

  std::cout << "The intersection:" << std::endl;
  for (it = intR.begin(); it != intR.end(); ++it) {
    std::cout << "--> ";
    print_polygon_with_holes (*it);
  }

  return 0;
}
```

File: examples/Boolean_set_operations_2/simple_join_intersect.cpp

Operations on Polygons with Holes

We have demonstrated that free functions that perform boolean set operations on simple polygons may output polygons with holes. In addition to these functions, the Boolean set-operations package provides the following overridden free functions that accept *General_polygon_with_holes_2* objects as their input - *complement()*, *intersection()*, *join()*, *difference()*, *symmetric_difference()* and *do_intersect()* The prototypes of most functions is the same as of their simpler counterparts that operate on simple polygons. The only exception is *complement(P, oi)*, which outputs a range of polygons with holes that represents the complement of the polygon with holes *P*.

The following example demonstrates how to compute the symmetric difference between two sets that contain holes. Each set is a rectangle that contains a rectangular hole in its interior, such that the symmetric difference between the two sets is a single polygon that contains of five holes:



```
/*! \file symmetric_difference.cpp
 * Computing the symmetric difference of two polygons with holes.
 */

#include <CGAL/Exact_predicates_exact_constructions_kernel.h>
#include <CGAL/Boolean_set_operations_2.h>
#include <list>


typedef CGAL::Exact_predicates_exact_constructions_kernel Kernel;
```

```
typedef Kernel::Point_2                                     Point_2;
typedef CGAL::Polygon_2<Kernel>                             Polygon_2;
typedef CGAL::Polygon_with_holes_2<Kernel>
   Polygon_with_holes_2;
typedef std::list<Polygon_with_holes_2>                     Pwh_list_2;

#include "print_utils.h"

int main ()
{
  // Construct P - a bounded rectangle that contains a rectangular hole.
  Polygon_2 outP;
  Polygon_2 holesP[1];

  outP.push_back (Point_2 (-3, -5));  outP.push_back (Point_2 (3, -5));
  outP.push_back (Point_2 (3, 5));    outP.push_back (Point_2 (-3, 5));
  holesP[0].push_back (Point_2 (-1, -3));
  holesP[0].push_back (Point_2 (-1, 3));
  holesP[0].push_back (Point_2 (1, 3));
  holesP[0].push_back (Point_2 (1, -3));

  Polygon_with_holes_2  P (outP, holesP, holesP + 1);
  std::cout << "P = "; print_polygon_with_holes (P);

  // Construct Q - a bounded rectangle that contains a rectangular hole.
  Polygon_2 outQ;
  Polygon_2 holesQ[1];

  outQ.push_back (Point_2 (-5, -3));  outQ.push_back (Point_2 (5, -3));
  outQ.push_back (Point_2 (5, 3));    outQ.push_back (Point_2 (-5, 3));
  holesQ[0].push_back (Point_2 (-3, -1));
  holesQ[0].push_back (Point_2 (-3, 1));
  holesQ[0].push_back (Point_2 (3, 1));
  holesQ[0].push_back (Point_2 (3, -1));

  Polygon_with_holes_2  Q (outQ, holesQ, holesQ + 1);
  std::cout << "Q = "; print_polygon_with_holes (Q);

  // Compute the symmetric difference of P and Q.
  Pwh_list_2 symmR;
  Pwh_list_2::const_iterator it;

  CGAL::symmetric_difference (P, Q, std::back_inserter(symmR));

  std::cout << "The symmetric difference:" << std::endl;
  for (it = symmR.begin(); it != symmR.end(); ++it) {
    std::cout << "--> ";
    print_polygon_with_holes (*it);
  }

  return 0;
}
```

File: examples/Boolean_set_operations_2/symmetric_difference.cpp

In some cases it is convenient to connect the outer boundary of a polygon with holes with the holes inside it. The function *connect_holes()* accepts a polygon with holes, and connects the topmost vertex in each hole with the polygon feature located directly above it (a vertex or an edge of the outer boundary, or of another hole). It produces an output sequence of points that match the traversal of all vertices in the connected polygon (note that there are additional vertices, induced by the vertical walls).

```
/*! \file connect_polygon.cpp
 * Connecting a polygon with holes.
 */

#include <CGAL/Exact_predicates_exact_constructions_kernel.h>
#include <CGAL/connect_holes.h>
#include <list>

typedef CGAL::Exact_predicates_exact_constructions_kernel Kernel;
typedef Kernel::Point_2                                   Point_2;
typedef CGAL::Polygon_2<Kernel>                           Polygon_2;
typedef CGAL::Polygon_with_holes_2<Kernel>
   Polygon_with_holes_2;

int main (int argc, char* argv[])
{

  // Get the name of the input file from the command line, or use the
    default
  // pgn_holes.dat file if no command-line parameters are given.
  //more data files can be found under test data
  //boundary no other connections are made.
  const char* filename = (argc > 1) ? argv[1] : "pgn_holes.dat";
  std::ifstream input_file (filename);
  if (! input_file.is_open())
  {
    std::cerr << "Failed to open the " << filename <<std::endl;
    return -1;
  }

  // Read a polygon with holes from a file.
  Polygon_2               outerP;
  unsigned int            num_holes;

  input_file >> outerP;
  input_file >> num_holes;

  std::vector<Polygon_2>  holes (num_holes);
  unsigned int            k;

  for (k = 0; k < num_holes; k++)
    input_file >> holes[k];

  Polygon_with_holes_2    P (outerP, holes.begin(), holes.end());

  // Connect the outer boundary of the polygon with its holes.
```

```
  std::list<Point_2>            pts;
  std::list<Point_2>::iterator  pit;

  connect_holes (P, std::back_inserter (pts));

  for (pit = pts.begin(); pit != pts.end(); ++pit)
    std::cout << '(' << *pit << ")  ";
  std::cout << std::endl;

  return (0);
}
```

File: examples/Boolean_set_operations_2/connect_polygon.cpp

19.3.2 Operating on Polygon Sets

We argue that the result of a regularized operations on two polygons (or polygons with holes) P and Q is typically a collection of several disconnected polygons with holes. It is only natural to represent such a collection in terms of a class, making it possible to operate on the set resulting from computing, for example, $P \setminus Q$.

A central component in the Boolean set-operations package is the *Polygon_set_2<Kernel, Container, Dcel>* class-template. An instance of this class represents a point set formed by the collection of several disconnected polygons with holes. It employs the *Arrangement_2* class to represent this point set in the plane as a planar arrangement; see Chapter 32. The instantiated *Dcel* type is used to represent the underlying internal arrangement. It must model the concept *GeneralPolygonSetDcel*, and defaults to *Gps_default_dcel*. You can override this default, with a different DCEL class, typically an extension of the default. Overriding the default is necessary only if you intend to obtain the underlying internal arrangement and process it further.

An instance S of a *Polygon_set_2* class usually represents the result of a sequence of operations that were applied on some input polygons. The representation of S is unique, regardless of the particular sequence of operations that were applied in order to arrive at it.

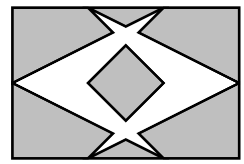
In addition, a polygon-set object can be constructed from a single polygon object or from a polygon-with-holes object. Once constructed, it is possible to insert new polygons (or polygons with holes) into the set using the *insert()* method, as long as the inserted polygons and the existing polygons in the set are disjoint. The *Polygon_set_2* class also provides access functions for accessing the polygons with holes it contains, and a few queries. The most important query is *S.oriented_side(q)*, which determined whether the query point q is contained in the interior of the set S, lies on the boundary of the set, or neither.

The *General_polygon_set_2* class defines the predicate *do_intersect()* and the methods *complement()*, *intersection()*, *join()*, *difference()* and *symmetric_difference()* as member functions. The operands to these functions may be simple polygons (*Polygon_2* object), polygons with holes (*General_polygon_with_holes_2* objects), or polygon sets (*General_polygon_set_2* objects). Thus, each of the function mentioned above is actually realized by a set overriding member functions.

Member functions of the *General_polygon_set_2* that perform Boolean set-operations come in two flavors: for example, *S.join(P, Q)* computes the union of P and Q and assigned the result to S, while *S.join(P)* performs the operation $S \longleftarrow S \cup P$. Similarly, *S.complement(P)* sets S to be the complement of P, while $S.complement()$ simply negates the set S.

A Sequence of Set Operations

The free functions reviewed in Section 19.3.1 serve as a wrapper for the polygon-set class, and are only provided for convenience. A typical such function constructs a pair of *General_polygon_set_2* objects, invokes the appropriate method to apply the desired Boolean operation, and transforms the resulting polygon set to the required output format. Thus, when several operations are performed in a sequence, it is much more efficient to use the member functions of the *General_polygon_set_2* type directly, as the extraction of the polygons from the internal representation for some operation, and the reconstruction of the internal representation for the succeeding operation could be time consuming.



The next example performs a sequence of three Boolean set-operations. First, it computes the union of two simple polygons depicted in Figure 19.3.1 (a). It then computes the complement of the result of the union operation. Finally, it computes the intersection of the result of the complement operation with a rectangle, confining the final result to the area of the rectangle. The resulting set S is comprised of two components: a polygon with a hole, and a simple polygon contained in the interior of this hole.

```
/*! \file sequence.cpp
 * Performing a sequence of Boolean set-operations.
 */

#include <CGAL/Exact_predicates_exact_constructions_kernel.h>
#include <CGAL/Polygon_2.h>
#include <CGAL/Polygon_with_holes_2.h>
#include <CGAL/Polygon_set_2.h>

#include <list>

typedef CGAL::Exact_predicates_exact_constructions_kernel Kernel;
typedef Kernel::Point_2                                   Point_2;
typedef CGAL::Polygon_2<Kernel>                           Polygon_2;
typedef CGAL::Polygon_with_holes_2<Kernel>
    Polygon_with_holes_2;
typedef CGAL::Polygon_set_2<Kernel>                       Polygon_set_2;

#include "print_utils.h"

int main ()
{
  // Construct the two initial polygons and the clipping rectangle.
  Polygon_2 P;
  P.push_back (Point_2 (0, 1));
  P.push_back (Point_2 (2, 0));
  P.push_back (Point_2 (1, 1));
  P.push_back (Point_2 (2, 2));

  Polygon_2 Q;
  Q.push_back (Point_2 (3, 1));
  Q.push_back (Point_2 (1, 2));
  Q.push_back (Point_2 (2, 1));
  Q.push_back (Point_2 (1, 0));

  Polygon_2 rect;
  rect.push_back (Point_2 (0, 0));
  rect.push_back (Point_2 (3, 0));
```

```
  rect.push_back (Point_2 (3, 2));
  rect.push_back (Point_2 (0, 2));

  // Perform a sequence of operations.
  Polygon_set_2 S;
  S.insert (P);
  S.join (Q);                    // Compute the union of P and Q.
  S.complement();                // Compute the complement.
  S.intersection (rect);         // Intersect with the clipping rectangle.

  // Print the result.
  std::list<Polygon_with_holes_2> res;
  std::list<Polygon_with_holes_2>::const_iterator it;

  std::cout << "The result contains " <<
   S.number_of_polygons_with_holes()
            << " components:" << std::endl;

  S.polygons_with_holes (std::back_inserter (res));
  for (it = res.begin(); it != res.end(); ++it) {
    std::cout << "--> ";
    print_polygon_with_holes (*it);
  }

  return 0;
}
```

File: examples/Boolean_set_operations_2/sequence.cpp

Inserting Non Intersecting Polygons

If you want to compute the union of a polygon P (P may be a simple polygon or a polygon-with-holes object) with a general-polygon set R, and store the result in R, you can construct a polygon set $S(P)$, and apply the *union* operation as follows:

```
General_polygon_2 S (P);
R.join (S);
```

As a matter of fact, you can apply the union operation directly:

```
R.join (P);
```

However, if you know that the polygon does not intersect any one of the polygons represented by R, you can use the more efficient method *insert()*:

```
R.insert (P);
```

As *insert()* assumes that $P \cap R = \emptyset$, it does not try to compute intersections between the boundaries of P and of R. This fact significantly speeds up the insertion process in comparison with the insertion of a non-disjoint polygon that intersects R.

The *insert()* function is also overloaded, so it can also accept a range of polygons. When a range of polygons are inserted into a polygon set S, all the polygons in the range and the polygons represented by S must be pairwise disjoint in their interiors.

19.3.3 Performing Aggregated Operations

There are a few options to compute the union of a set of polygons $P_1, \ldots P_m$. You can do it incrementally as follows. At each step compute the union of $S_{k-1} = \bigcup_{i=1}^{k-1} P_i$ with P_k and obtain S_k. Namely, if the polygon set is given as a pair of iterator *[begin, end)*, the following loop computes their union in S.

```
InputIterator iter = begin;
Polygon_set_2 S (*iter);

while (++iter != end) {
  S.join (*iter);
  ++iter;
}
```

A second option is to use a divide-and-conquer approach. You bisect the set of polygons into two sets. Compute the union of each set recursively and obtain the partial results in S_1 and S_2, and finally, you compute the union $S_1 \cup S_2$. However, the union operation can be done more efficiently for sparse polygons, having a relatively small number of intersections, using a third option that simultaneously computes the union of all polygons. This is done by constructing a planar arrangement of all input polygons, utilizing the sweep-line algorithm, then extracting the result from the arrangement. Similarly, it is also possible to aggregately compute the intersection $\bigcap_{i=1}^{m} P_i$ of a set of input polygons.

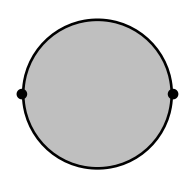
Our package provides the free overloaded functions *join()* and *intersect()* that aggregately compute the union and the intersection of a range of input polygons. There is no restriction on the polygons in the range — naturally, they may intersect each other. The package provides the overloaded free function *do_intersect(begin, end)* that determines whether the polygons in the range defined by the two input iterators *[begin, end)* intersect.

The class *General_polygon_set_2* also provides equivalent member functions that aggragately operate on a range of input polygons or polygons with holes. When such a member function is called, the general polygons represented by the current object are considered operands as well. Thus, you can easily compute the union of our polygon range as follows:

```
Polygon_set_2 S;
S.join (begin, end);
```

19.4 Boolean Set-Operations on General Polygons

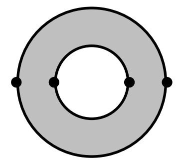
In previous sections only ordinary (linear) polygons were dealt with. Namely, closed point sets bounded by piecewise linear curves. The Boolean set-operations package allows a more general geometric mapping of the polygon edges. The operations provided by the package operate on point sets bounded by x-monotone segments of general curves (e.g., conic arcs and segments of polynomial functions). For example, the point set depicted on the right is a general polygon bounded by two x-monotone circular arcs that correspond to the lower half and the upper half of a circle, respectively.



Using the topological terminology, a general polygon can represent any simply-connected point set whose boundary is a strictly simple curve. Such a polygon is a model of the *GeneralPolygon_2* concept. A model of this concept must fulfill the following requirements:

- A general polygon is constructible from a range of pairwise interior disjoint x-monotone curves $c_1, \ldots, c_n$. The target point of the kth curve c_k and the source point of the next curve in the range (in a cyclic order) must coincide, such that this point defines the kth polygon vertex.
- It is possible to traverse the x-monotone curves that form the edges of a general polygon.

The concept *GeneralPolygonWithHoles_2* is defined in an analogous way to the definition of linear polygons with holes. A model of this concept represent a bounded or an unbounded set that may not be simply connected, and must provide the following operations:



- Construction for a general polygon that represent the outer boundary and a range of general polygons that represent the holes.
- Accessing the general polygons that represents the outer boundary (in case of a bounded set).
- Traversing the holes.

In Section 19.3 we introduce the classes *Polygon_2* and *Polygon_with_holes_2* that model the concepts *GeneralPolygon_2* and *GeneralPolygonWithHoles_2* respectively. In this section we introduce other models of these two concepts.

The central class-template *General_polygon_set_2<Traits,Dcel>* is used to represent point sets that are comprised of a finite number of pairwise disjoint general polygons with holes, and provides various Boolean set-operations on such sets. It is parameterized by a *traits* class and a DCEL class. The former defines the type of points used to represent polygon vertices and the type of x-monotone curves that represent the polygon edges. The traits class also provides primitive geometric operations that operate on objects of these types. The DCEL class is used to represent the underlying internal *Arrangement_2* data structure. The instantiated *Dcel* type is used to represent the underlying internal arrangement. It must model the concept *GeneralPolygonSetDcel*, and defaults to *Gps_default_dcel*. You can override this default, with a different DCEL class, typically an extension of the default. Overriding the default is necessary only if you intend to obtain the underlying internal arrangement and process it further.

An instantiated *General_polygon_set_2* class defines the nested types *General_polygon_set_2<Traits,Dcel> ::Polygon_2* and *General_polygon_set_2<Traits,Dcel>::Polygon_with_holes_2*, which model the concepts *GeneralPolygon_2* and *GeneralPolygonWithHoles_2* respectively.

19.4.1 The Traits-Class Concepts

The traits class used to instantiate the *General_polygon_set_2* class template must model the concept *GeneralPolygonSetTraits_2*, and is tailored to handle a specific family of curves. The concept *GeneralPolygonSetTraits_2* refines the concept *ArrangementDirectionalXMonotoneTraits_2* specified next.

The concept *ArrangementDirectionalXMonotoneTraits_2* refines the concept *ArrangementXMonotoneTraits_2* (see Section 32.4.1 in the 2D Arrangements chapter). Thus, a model of this concept must define the type *X_monotone_curve_2*, which represents an x-monotone curve, and the type *Point_2*, with represents a planar point. Such a point may be an endpoint of an x-monotone curve or an intersection point between two curves.

It must also provide various geometric predicates and operations on these types listed by the base concept, such as determining whether a point lies above or below an x-monotone curve, computing the intersections between two curves, etc. Note that the base concept does not assume that x-monotone curves are directed: an x-monotone curve is not required to have a designated *source* and *target*, it is only required to determine the left (lexicographically smaller) and the right (lexicographically larger) endpoints of a given curve.

The *ArrangementDirectionalXMonotoneTraits_2* concept treats its x-monotone curves as directed objects. It thus requires two additional operations on x-monotone curves:

- Given an x-monotone curve, compare its source and target points lexicographically.
- Given an x-monotone curve, construct its opposite curve (namely, swap its source and target points).

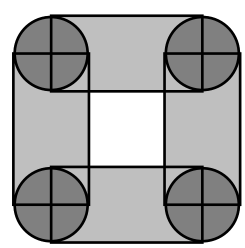
The traits classes *Arr_segment_traits_2*, *Arr_non_caching_segment_traits*, *Arr_circle_segment_traits_2*, *Arr_conic_traits_2* and *Arr_rational_arc_traits_2*, which are bundled in the *Arrangement_2* package and distributed with CGAL, are all models of the refined concept *ArrangementDirectionalXMonotoneTraits_2*.[3]

Just as with the case of computations using models of the *ArrangementXMonotoneTraits_2* concept, operations are robust only when exact arithmetic is used. When inexact arithmetic is used, (nearly) degenerate configurations may result in abnormal termination of the program or even incorrect results.

19.4.2 Operating on Polygons with Circular Arcs

Two traits classes are distributed with CGAL. The first one is named *Gps_segment_traits_2*, and it is used to perform Boolean set-operations on ordinary polygons and polygons with holes. In fact, the class *Polygon_set_2* introduced in Section 19.3.2 is a specialization of *General_polygon_set_2*<*Gps_segment_traits_2*>. This class defined its polygon and polygon with holes types, such that the usage of this traits class is encapsulated in the polygon-set class.

The other predefined traits class is named *Gps_circle_segment_traits_2*<*Kernel*> and is parameterized by a geometric CGAL kernel. By instantiating the *General_polygon_set_2* with this traits class, we obtain the representation of a polygon whose boundary may be comprised of line segments and circular arcs. The circle–segment traits class provides predicates and constructions on non-linear objects; yet, it uses only rational arithmetic and is very efficient as a consequence.



The following example uses the *Gps_circle_segment_traits_2* class to compute the union of four rectangles and four circles. Each circle is represented as a general polygon having two x-monotone circular arcs. The union is computed incrementally, resulting with a single polygon with a single hole, as depicted on the right. Note that as the four circles are disjoint, their union is computed with the *insert* method, while the union with the rectangles is computed with the *join* operator.

```
/*! \file circle_segment.cpp
 * Handling circles and linear segments concurrently.
 */

#include <CGAL/Exact_predicates_exact_constructions_kernel.h>
#include <CGAL/Gps_circle_segment_traits_2.h>
```

[3] The *Arr_polyline_traits_2* class is *not* a model of the, *ArrangementDirectionalXMonotoneTraits_2* concept, as the x-monotone curve it defines is always directed from left to right. Thus, an opposite curve cannot be constructed. However, it is not very useful to construct a polygon whose edges are polylines, as an ordinary polygon with linear edges can represent the same entity.

```
#include <CGAL/General_polygon_set_2.h>
#include <CGAL/Lazy_exact_nt.h>

#include <list>

typedef CGAL::Exact_predicates_exact_constructions_kernel Kernel;
typedef Kernel::Point_2                                   Point_2;
typedef Kernel::Circle_2                                  Circle_2;
typedef CGAL::Gps_circle_segment_traits_2<Kernel>         Traits_2;

typedef CGAL::General_polygon_set_2<Traits_2>             Polygon_set_2;
typedef Traits_2::General_polygon_2                       Polygon_2;
typedef Traits_2::General_polygon_with_holes_2
   Polygon_with_holes_2;
typedef Traits_2::Curve_2                                 Curve_2;
typedef Traits_2::X_monotone_curve_2
   X_monotone_curve_2;

// Construct a polygon from a circle.
Polygon_2 construct_polygon (const Circle_2& circle)
{
  // Subdivide the circle into two x-monotone arcs.
  Traits_2 traits;
  Curve_2 curve (circle);
  std::list<CGAL::Object>  objects;
  traits.make_x_monotone_2_object() (curve, std::back_inserter(objects));
  CGAL_assertion (objects.size() == 2);

  // Construct the polygon.
  Polygon_2 pgn;
  X_monotone_curve_2 arc;
  std::list<CGAL::Object>::iterator iter;

  for (iter = objects.begin(); iter != objects.end(); ++iter) {
    CGAL::assign (arc, *iter);
    pgn.push_back (arc);
  }

  return pgn;
}

// Construct a point from a rectangle.
Polygon_2 construct_polygon (const Point_2& p1, const Point_2& p2,
                             const Point_2& p3, const Point_2& p4)
{
  Polygon_2 pgn;
  X_monotone_curve_2 s1(p1, p2);    pgn.push_back(s1);
  X_monotone_curve_2 s2(p2, p3);    pgn.push_back(s2);
  X_monotone_curve_2 s3(p3, p4);    pgn.push_back(s3);
  X_monotone_curve_2 s4(p4, p1);    pgn.push_back(s4);
  return pgn;
}

// The main program:
```

```
int main ()
{
  // Insert four non-intersecting circles.
  Polygon_set_2 S;
  Polygon_2 circ1, circ2, circ3, circ4;

  circ1 = construct_polygon(Circle_2(Point_2(1, 1), 1));
   S.insert(circ1);
  circ2 = construct_polygon(Circle_2(Point_2(5, 1), 1));
   S.insert(circ2);
  circ3 = construct_polygon(Circle_2(Point_2(5, 5), 1));
   S.insert(circ3);
  circ4 = construct_polygon(Circle_2(Point_2(1, 5), 1));
   S.insert(circ4);

  // Compute the union with four rectangles incrementally.
  Polygon_2 rect1, rect2, rect3, rect4;

  rect1 = construct_polygon(Point_2(1, 0), Point_2(5, 0),
                            Point_2(5, 2), Point_2(1, 2));
  S.join (rect1);

  rect2 = construct_polygon(Point_2(1, 4), Point_2(5, 4),
                            Point_2(5, 6), Point_2(1, 6));
  S.join (rect2);

  rect3 = construct_polygon(Point_2(0, 1), Point_2(2, 1),
                            Point_2(2, 5), Point_2(0, 5));
  S.join (rect3);

  rect4 = construct_polygon(Point_2(4, 1), Point_2(6, 1),
                            Point_2(6, 5), Point_2(4, 5));
  S.join (rect4);

  // Print the output.
  std::list<Polygon_with_holes_2> res;
  S.polygons_with_holes (std::back_inserter (res));

  std::copy (res.begin(), res.end(),
             std::ostream_iterator<Polygon_with_holes_2>(std::cout,
   "\n"));
  std::cout << std::endl;

  return 0;
}
```

File: examples/Boolean_set_operations_2/circle_segment.cpp

19.4.3 General Polygon Set Traits Adapter

The concept *GeneralPolygon_2* and its generic model *General_polygon_2<ArrDirectionalXMonotoneTraits>* facilitate the production of general-polygon set traits classes. A model of the concept *GeneralPolygon_2* represents a simple point-set in the plane bounded by x-monotone curves. As opposed to the plain *Traits::Polygon_2* type defined by any traits class, it must define the type *X_monotone_curve_2*, which represents an x-monotone curve of the point-set boundary. It must provide a constructor from a range of such curves, and a pair of methods, namely *curves_begin()* and *curves_end()*, that can be used to iterate over the point-set boundary curves.

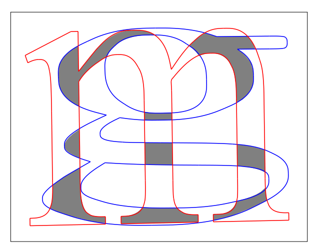
The class-template *General_polygon_2<ArrDirectionalXMonotoneTraits>* models the concept *GeneralPolygon_2*. Its sole template parameter must be instantiated with a model of the concept *ArrangementDirectionalXMonotoneTraits_2* from which it obtains the *X_monotone_curve_2* type. It uses the geometric operations on this type provided by such a model to maintain a container of directed curves of type *X_monotone_curve_2*, which represents a boundary of the general polygon.

The class-template *Gps_traits_2<ArrDirectionalXMonotoneTraits,GeneralPolygon>* models the concept *GeneralPolygonSetTraits_2*, and can be used to instantiate the class template *General_polygon_set_2*. It serves as an adapter for a geometric traits class, which models the concept *ArrangementDirectionalXMonotoneTraits_2*. It can be used for performing set-operations on general polygons. The implementation of the adapter is rather simple, as it is derived from the instantiated template-parameter *ArrXMonotoneTraits_2* inheriting its necessary types and methods. It further exploits the methods provided by the instantiated parameter *GeneralPolygon*, which is a model of the concept *GeneralPolygon_2*. By default, the *GeneralPolygon* parameter is defined as *General_polygon_2<ArrangementDirectionalXMonotoneTraits_2>*.

The code excerpt listed below defines a general-polygon set type that can be used to perform Boolean set-operations on point sets bounded by the x-monotone curve type defined by the arrangement-traits class *Arr_traits_2*, which is some representative model of the concept *ArrangementDirectionalXMonotoneTraits_2*.

```
#include <CGAL/General_polygon_2.h>
#include <CGAL/Gps_traits_2.h>

typedef CGAL::General_polygon_2<Arr_traits_2>               General_polygon_2;
typedef CGAL::Gps_traits_2<Arr_traits_2, General_polygon_2> Traits_2;
typedef CGAL::General_polygon_set_2<Traits_2>               General_polygon_set_2;
```

Instantiating the arrangement-traits *Arr_traits_2* above with the traits class that handle Bézier curves *Arr_bezier_traits_2*, results with the definition of a general-polygon set type that can be used to perform Boolean set-operations on point sets bounded by Bézier curves.
The next example computes the intersection of two general polygons bounded by Bézier curves read from two input files respectively. The default input files our example uses (*char_g.dat* and *char_m.dat*) define two general polygons shaped in the form of the characters **g** and **m** in the Times New Roman font respectively. Their intersection comprises nine simple polygons, as depicted to the right.



Recall that every Bézier curve is defined by a sequence of control points that form chains (see Section 32.6.7. The last control point of every curve must be identical to the first control point of its successor. The function *read_Bezier_polygon()* included in the example reads the curves from an input file until they form a closed chain, which is assumed to be the outer boundary of the polygon. If more curves are available, its starts constructing polygons that correspond to holes in the area bounded by the outer boundary. Note that this function is also responsible for subdividing the input Bézier curves into x-monotone subcurves, as required by the *Gps_traits_2* adapter.

```
/*! \file bezier_traits_adapter.cpp
 * Using the traits adaptor to generate a traits class for Bezier
   polygons.
 */
```

```
#include <CGAL/basic.h>

#ifndef CGAL_USE_CORE
#include <iostream>
int main ()
{
  std::cout << "Sorry, this example needs CORE ..." << std::endl;
  return (0);
}
#else

#include <CGAL/Cartesian.h>
#include <CGAL/CORE_algebraic_number_traits.h>
#include <CGAL/Arr_Bezier_curve_traits_2.h>
#include <CGAL/Arrangement_2.h>
#include <CGAL/Gps_traits_2.h>
#include <CGAL/Boolean_set_operations_2.h>
#include <CGAL/Timer.h>

#include <fstream>

typedef CGAL::CORE_algebraic_number_traits              Nt_traits;
typedef Nt_traits::Rational                             Rational;
typedef Nt_traits::Algebraic                            Algebraic;

typedef CGAL::Cartesian<Rational>                       Rat_kernel;
typedef CGAL::Cartesian<Algebraic>                      Alg_kernel;
typedef CGAL::Arr_Bezier_curve_traits_2<Rat_kernel, Alg_kernel,
   Nt_traits>
                                                        Traits_2;

typedef Rat_kernel::Point_2                             Rat_point_2;
typedef Traits_2::Curve_2                               Bezier_curve_2;
typedef Traits_2::X_monotone_curve_2
   X_monotone_curve_2;
typedef CGAL::Gps_traits_2<Traits_2>                    Gps_traits_2;
typedef Gps_traits_2::General_polygon_2                 Polygon_2;
typedef Gps_traits_2::General_polygon_with_holes_2
   Polygon_with_holes_2;
typedef std::list<Polygon_with_holes_2>                 Polygon_set;

/*! Read a general polygon with holes, formed by Bezier curves, from the
 * given input file.
 */
bool read_Bezier_polygon (const char* filename, Polygon_with_holes_2& P)
{
  // Open the input file.
  std::ifstream   in_file (filename);

  if (! in_file.is_open())
    return false;

  // Read the number of curves.
  unsigned int      n_curves;
```

```
unsigned int      k;

in_file >> n_curves;

// Read the curves one by one, and construct the general polygon these
// curve form (the outer boundary and the holes inside it).
Traits_2                    traits;
Traits_2::Make_x_monotone_2 mk_x_monotone =
 traits.make_x_monotone_2_object();
bool                        first = true;
Rat_point_2                 p_0;
std::list<X_monotone_curve_2>  xcvs;
Rat_kernel                  ker;
Rat_kernel::Equal_2         equal = ker.equal_2_object();
std::list<Polygon_2>        pgns;

for (k = 0; k < n_curves; k++) {
  // Read the current curve and subdivide it into x-monotone subcurves.
  Bezier_curve_2                          B;
  std::list<CGAL::Object>                 x_objs;
  std::list<CGAL::Object>::const_iterator xoit;
  X_monotone_curve_2                      xcv;

  in_file >> B;
  mk_x_monotone (B, std::back_inserter (x_objs));

  for (xoit = x_objs.begin(); xoit != x_objs.end(); ++xoit) {
    if (CGAL::assign (xcv, *xoit))
      xcvs.push_back (xcv);
  }

  // Check if the current curve closes a polygon, namely whether it
 target
  // point (the last control point) equals the source of the first
 curve in
  // the current chain.
  if (! first) {
    if (equal (p_0, B.control_point(B.number_of_control_points() -
 1))) {
      // Push a new polygon into the polygon list. Make sure that the
 polygon
      // is counterclockwise oriented if it represents the outer
 boundary
      // and clockwise oriented if it represents a hole.
      Polygon_2          pgn (xcvs.begin(), xcvs.end());
      CGAL::Orientation  orient = pgn.orientation();

      if ((pgns.empty() && orient == CGAL::CLOCKWISE) ||
          (! pgns.empty() && orient == CGAL::COUNTERCLOCKWISE))
        pgn.reverse_orientation();

      pgns.push_back (pgn);
      xcvs.clear();
      first = true;
```

```
      }
    }
    else {
      // This is the first curve in the chain - store its source point.
      p_0 = B.control_point(0);
      first = false;
    }
  }

  if (! xcvs.empty())
    return false;

  // Construct the polygon with holes.
  std::list<Polygon_2>::iterator     pit = pgns.begin();

  ++pit;
  P = Polygon_with_holes_2 (pgns.front(), pit, pgns.end());

  return true;
}

// The main program.
int main (int argc, char* argv[])
{
  // Get the name of the input files from the command line, or use the
    default
  // char_g.dat and char_m.dat files if no command-line parameters are
    given.
  const char* filename1 = (argc > 1) ? argv[1] : "char_g.dat";
  const char* filename2 = (argc > 2) ? argv[2] : "char_m.dat";

  // Read the general polygons from the input files.
  CGAL::Timer           timer;
  Polygon_with_holes_2  P1, P2;

  timer.start();

  if (! read_Bezier_polygon (filename1, P1)) {
    std::cerr << "Failed to read " << filename1 << " ..." << std::endl;
    return 1;
  }

  if (! read_Bezier_polygon (filename2, P2)) {
    std::cerr << "Failed to read " << filename2 << " ..." << std::endl;
    return 1;
  }

  timer.stop();
  std::cout << "Constructed the input polygons in " << timer.time()
            << " seconds." << std::endl << std::endl;

  // Compute the intersection of the two polygons.
  Polygon_set                     R;
  Polygon_set::const_iterator     rit;
```

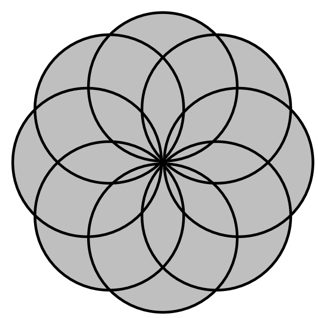
```
  timer.reset();
  timer.start();
  CGAL::intersection (P1, P2, std::back_inserter(R));
  timer.stop();

  std::cout << "The intersection polygons are of sizes: {";
  for (rit = R.begin(); rit != R.end(); ++rit)
    std::cout << ' ' << rit->outer_boundary().size();
  std::cout << " }" << std::endl;
  std::cout << "The intersection computation took "
            << timer.time() << " seconds." << std::endl;

  return 0;
}

#endif
```

File: examples/Boolean_set_operations_2/bezier_traits_adapter.cpp

19.4.4 Example - Aggregated Operations

In Section 19.3.3 we describe how aggregated union and intersection operations can be applied to a collection of ordinary polygons or polygons with holes. Naturally, the aggregated operations can be applied also to collections of general polygons. As was the case with ordinary polygons, using aggregated operations is recommended when the number of intersections of the input polygons is of the same order of magnitude as the complexity of the result. If this is not the case, computing the result incrementally may prove faster.

The next example computes the union of eight unit discs whose centers are placed a unit distance from the origin, as depicted to the right. The example also allows users to provide a different number of discs through the command line.



```
/*! \file set_union.cpp
 * Computing the union of a set of circles.
 */

#include <CGAL/Exact_predicates_exact_constructions_kernel.h>
#include <CGAL/Gps_circle_segment_traits_2.h>
#include <CGAL/Boolean_set_operations_2.h>
#include <CGAL/Lazy_exact_nt.h>

#include <list>
#include <cstdlib>
#include <cmath>
```

```
typedef CGAL::Exact_predicates_exact_constructions_kernel Kernel;
typedef Kernel::Point_2                                   Point_2;
typedef Kernel::Circle_2                                  Circle_2;
typedef CGAL::Gps_circle_segment_traits_2<Kernel>         Traits_2;

typedef CGAL::General_polygon_set_2<Traits_2>             Polygon_set_2;
typedef Traits_2::Polygon_2                               Polygon_2;
typedef Traits_2::Polygon_with_holes_2
   Polygon_with_holes_2;
typedef Traits_2::Curve_2                                 Curve_2;
typedef Traits_2::X_monotone_curve_2
   X_monotone_curve_2;

// Construct a polygon from a circle.
Polygon_2 construct_polygon (const Circle_2& circle)
{
  // Subdivide the circle into two x-monotone arcs.
  Traits_2 traits;
  Curve_2 curve (circle);
  std::list<CGAL::Object> objects;
  traits.make_x_monotone_2_object() (curve, std::back_inserter(objects));
  CGAL_assertion (objects.size() == 2);

  // Construct the polygon.
  Polygon_2 pgn;
  X_monotone_curve_2 arc;
  std::list<CGAL::Object>::iterator iter;

  for (iter = objects.begin(); iter != objects.end(); ++iter) {
    CGAL::assign (arc, *iter);
    pgn.push_back (arc);
  }

  return pgn;
}

// The main program:
int main (int argc, char* argv[])
{
  // Read the number of circles from the command line.
  unsigned int n_circles = 8;
  if (argc > 1) n_circles = std::atoi(argv[1]);

  // Create the circles, equally spaced of the circle x^2 + y^2 = 1.
  const double pi = std::atan(1.0) * 4;
  const double n_circles_reciep = 1.0 / n_circles;
  const double radius = 1;
  const double frac = 2 * pi * n_circles_reciep;
  std::list<Polygon_2> circles;
  unsigned int k;
  for (k = 0; k < n_circles; k++) {
    const double angle = frac * k;
    const double x = radius * std::sin(angle);
    const double y = radius * std::cos(angle);
```

```
    Point_2 center = Point_2(x, y);
    Circle_2 circle(center, radius);

    circles.push_back (construct_polygon (circle));
  }

  // Compute the union aggragately.
  std::list<Polygon_with_holes_2> res;
  CGAL::join (circles.begin(), circles.end(), std::back_inserter (res));

  // Print the result.
  std::copy (res.begin(), res.end(),
             std::ostream_iterator<Polygon_with_holes_2>(std::cout,
   "\n"));
  std::cout << std::endl;

  return 0;
}
```

File: examples/Boolean_set_operations_2/set_union.cpp

2D Regularized Boolean Set-Operations Reference Manual

Efi Fogel, Ophir Setter, Ron Wein, Guy Zucker, Baruch Zukerman, and Dan Halperin

Introduction

This package consists of the implementation of Boolean set-operations on point sets bounded by x-monotone curves in 2-dimensional Euclidean space. In particular, it contains the implementation of regularized Boolean set-operations, intersection predicates, and point containment predicates.

19.5 Classified Reference Pages

Concepts

Classes

Functions

19.6 Alphabetical List of Reference Pages

CGAL::General_polygon_set_2<Traits,Dcel>

Class

Definition

An object of the *General_polygon_set_2<Traits,Dcel>* class-template represents a point set in the plane bounded by x monotone curves. Points in the set lie on the boundary or on the positive side of the curves. This class template provides methods to apply regularized Boolean set-operations and few other utility methods. An *Arrangement_2* data structure is internally used to represent the point set. The arrangement is represented as a doubly-connected edge-list (DCEL).

The *Traits* template-parameter should be instantiated with a model of the concept *GeneralPolygonSetTraits_2*. The traits class defines the types of points, x-monotone curves, general polygons, and general polygons with holes, that is *Traits::Point_2*, *Traits::X_monotone_curve_2*, *Traits::Polygon_2*, and *Traits::Polygon_with_holes_2*, respectively. *Traits::Point_2* must be the type of the endpoints of *Traits::X_monotone_curve_2*, and *Traits::X_monotone_curve_2* must be the type of the curves that comprise the boundaries of the general polygons. The traits class supports geometric operations on the types above. We sometimes use the term *polygon* instead of general polygon for simplicity hereafter.

The template parameter *Dcel* should be instantiated with a model of the concept *GeneralPolygonSetDcel*. It is instantiated by default with the type *Gps_default_dcel<Traits>*. You can override this default, with a different DCEL class, typically an extension of the *Gps_default_dcel* class template. Overriding the default is necessary only if you intend to obtain the underlying internal arrangement and process it further.

The input and output of the Boolean set-operations methods consist of one or more general polygons, some of which may have holes. In particular, these methods operate on pairs of objects of type *General_polygon_set_2<Traits,Dcel>*, or directly on objects of type *Traits::Polygon_2* or *Traits::Polygon_with_holes_2*. An object of type *Traits::Polygon_2* is a valid operand, only if it is simple and its boundary is oriented counterclockwise. An object of type *Traits::Polygon_with_holes_2* is valid, only if its outer boundary bounds a relatively simple general polygon oriented counterclockwise, and each one of its holes is a simple polygon oriented clockwise. The holes are pairwise disjoint, except perhaps at the vertices, and are contained in the polygon bounded by the outer boundary. The outer boundary and the holes are also pairwise disjoint, except perhaps at the vertices.

Types

General_polygon_set_2<Traits,Dcel>:: Traits_2	the traits class in use.
General_polygon_set_2<Traits,Dcel>:: Polygon_2	the general polygon type. Must model the *GpsTraitsGeneralPolygon_2* Concept.
General_polygon_set_2<Traits,Dcel>:: Polygon_with_holes_2	the general polygon with holes type. Must model the *GpsTraitsGeneralPolygonWithHoles_2* Concept.
General_polygon_set_2<Traits,Dcel>:: Size	number of polygons with holes size type.
General_polygon_set_2<Traits,Dcel>:: Arrangement_2	the arrangement type used internally.

Creation

General_polygon_set_2<Traits,Dcel> gps;	constructs an empty set of polygons represented by an empty arrangement.
General_polygon_set_2<Traits,Dcel> gps(Self other);	copy constructor.
General_polygon_set_2<Traits,Dcel> gps(Traits & traits);	constructs an empty set of polygons that uses the given *traits* instance for performing the geometric operations.
General_polygon_set_2<Traits,Dcel> gps(Polygon_2 pgn);	constructs a set of polygons that consists of the single polygon *pgn*.
General_polygon_set_2<Traits,Dcel> gps(Polygon_with_holes_2 pgn_with_holes);	constructs a set of polygons that consists of the single polygon with holes *pgn_with_holes*.

Access Functions

template <class OutputIterator>

OutputIterator	*gps.polygons_with_holes(OutputIterator out)*	obtains the polygons with holes represented by *gps*.
Size	*gps.number_of_polygons_with_holes() const*	returns the total number of general polygons represented by *gps*.
bool	*gps.is_empty() const*	returns *true* if *gps* represents an empty set.
bool	*gps.is_plane() const*	returns *true* if *gps* represents the entire plane.
Traits&	*gps.traits() const*	obtains an instance of the traits. If the traits was passed as a parameter to the constructor of *gps*, it is returned. Otherwise, a newly created instance is returned.
Arrangement_2	*gps.arrangement() const*	obtains the arrangement data structure that internally represents the general-polygon set.

Modifiers

void *gps.clear()*

clears *gps*.

void *gps.insert(Polygon_2 & pgn)*

inserts *pgn* into *gps*.
Precondition: *pgn* and the point set represented by *gps* are disjoint. This precondition enables the use of very efficient insertion methods. Use the respective overloaded method that inserts a polygon of type *Polygon_with_holes_2*, if only a relaxed precondition can be guaranteed. If even the relaxed precondition cannot be guaranteed, use the *join* method.

void *gps.insert(Polygon_with_holes_2 & pgn_with_holes)*

inserts *pgn_with_holes* into *gps*.
Precondition: *pgn_with_holes* does not intersect with the point set represented by *gps*, except maybe at the vertices. If this relaxed precondition cannot be guaranteed, use the *join* method.

template <class InputIterator>
void *gps.insert(InputIterator begin, InputIterator end)*

inserts the range of polygons (or polygons with holes) into *gps*. (The value type of the input iterator is used to distinguish between the two.)
Precondition: If the given range contains objects of type *Polygon_with_holes_2*, then these polygons and the point set represented by *gps* are pairwise disjoint, except maybe at the vertices. If the given range contains objects of type *Polygon_2*, then these polygons and the point set represented by *gps* are pairwise disjoint without any exceptions.

template <class InputIterator1, class InputIterator2>
void *gps.insert(InputIterator1 pgn_begin,*
InputIterator1 pgn_end,
InputIterator2 pgn_with_holes_begin,
InputIterator2 pgn_with_holes_end)

inserts the two ranges of polygons and polygons with holes into *gps*.
Precondition: All polygons in the first range, all polygon with holes in the second range, and the point set represented by *gps* are pairwise disjoint, except maybe at the vertices

void *gps.complement()*

computes the complement of *gps*.

void *gps.complement(Polygon_set_2 other)*

computes the complement of *other*. *gps* is overridden by the result.

Univariate Operations

In the following univariate and bivariate methods the result is placed in *gps* after it is cleared.

void gps.intersection(General_polygon_set_2 other)

computes the intersection of *gps* and *other*.

void gps.intersection(Polygon_2 pgn)

computes the intersection of *gps* and *pgn*.

void gps.intersection(Polygon_with_holes_2 pgn)

computes the intersection of *gps* and *pgn*.

template <class InputIterator>
void gps.intersection(InputIterator begin, InputIterator end)

computes the intersection of a collection of point sets. The collection consists of the polygons (or polygons with holes) in the given range and the point set represented by *gps*. (The value type of the input iterator is used to distinguish between the two options.)

template <class InputIterator1, class InputIterator2>
void gps.intersection(InputIterator1 pgn_begin, InputIterator1 pgn_end, InputIterator2 pgn_with_holes_begin, InputIterator2 pgn_with_holes_end)

computes the intersection of a collection of point sets. The collection consists of the polygons and polygons with holes in the given two ranges and the point set represented by *gps*.

void gps.join(General_polygon_set_2 other)

computes the union of *gps* and *other*.

void gps.join(Polygon_2 pgn)

computes the union of *gps* and *pgn*.

void gps.join(Polygon_with_holes_2 pgn)

computes the union of *gps* and *pgn*.

template <class InputIterator>
void gps.join(InputIterator begin, InputIterator end)

computes the union of the polygons (or polygons with holes) in the given range and the point set represented by *gps*. (The value type of the input iterator is used to distinguish between the two options.)

template <class InputIterator1, class InputIterator2>
void gps.join(InputIterator1 pgn_begin, InputIterator1 pgn_end, InputIterator2 pgn_with_holes_begin, InputIterator2 pgn_with_holes_end)

computes the union of the polygons and polygons with holes in the given two ranges and the point set represented by *gps*.

void *gps.difference(General_polygon_set_2 other)*

computes the difference between *gps* and *other*.

void *gps.difference(Polygon_2 pgn)* computes the difference between *gps* and *pgn*.

void *gps.difference(Polygon_with_holes_2 pgn)* computes the difference between *gps* and *pgn*.

void *gps.symmetric_difference(General_polygon_set_2 other)*

computes the symmetric difference between *gps* and *other*.

void *gps.symmetric_difference(Polygon_2 pgn)* computes the symmetric difference between *gps* and *pgn*.

void *gps.symmetric_difference(Polygon_with_holes_2 pgn)*

computes the symmetric difference between *gps* and *pgn*.

template <class InputIterator>
void *gps.symmetric_difference(InputIterator begin, InputIterator end)*

computes the symmetric difference (xor) of a collection of point sets. The collection consists of the polygons (or polygons with holes) in the given range and the point set represented by *gps*. (The value type of the input iterator is used to distinguish between the two options.)

template <class InputIterator1, class InputIterator2>
void *gps.symmetric_difference(InputIterator1 pgn_begin,*
InputIterator1 pgn_end,
InputIterator2 pgn_with_holes_begin,
InputIterator2 pgn_with_holes_end)

computes the symmetric difference (xor) of a collection of point sets. The collection consists of the polygons and polygons with holes in the given two ranges and the point set represented by *gps*.

Bivariate Operations

The following bivariate function replace *gps* with the result.

void *gps.intersection(General_polygon_set_2 gps1, General_polygon_set_2 gps2)*

computes the intersection of *gps1* and *gps2*.

void *gps.join(General_polygon_set_2 gps1, General_polygon_set_2 gps2)*

computes the union of *gps1* and *gps2*.

void *gps.difference(General_polygon_set_2 gps1, General_polygon_set_2 gps2)*

computes the difference between *gps1* and *gps2*.

void *gps.symmetric_difference(General_polygon_set_2 gps1, General_polygon_set_2 gps2)*

computes the symmetric difference between *gps1* and *gps2*.

Query Functions

bool	*gps.do_intersect(General_polygon_set_2 other)*	returns *true* if *gps* and *other* intersect in their interior, and *false* otherwise.
bool	*gps.do_intersect(Polygon_2 pgn)*	returns *true* if *gps* and *pgn* intersect in their interior, and *false* otherwise.
bool	*gps.do_intersect(Polygon_with_holes_2 pgn)*	returns *true* if *gps* and *pgn* intersect in their interior, and *false* otherwise.

template <class InputIterator>
void *gps.do_intersect(InputIterator begin, InputIterator end)*

returns *true* if the interior of the point sets in a collection intersect, and *false* otherwise. The collection consists of the polygons (or polygons with holes) in the given range and the point set represented by *gps*. (The value type of the input iterator is used to distinguish between the two options.)

template <class InputIterator1, class InputIterator2>
void *gps.do_intersect(InputIterator1 pgn_begin, InputIterator1 pgn_end, InputIterator2 pgn_with_holes_begin, InputIterator2 pgn_with_holes_end)*

returns *true* if the interior of the point sets in a collection intersect, and *false* otherwise. The collection consists of the polygons and polygons with holes in the given two ranges and the point set represented by *gps*.

bool *gps.locate(Point_2 p, Polygon_with_holes_2 & pgn)*

obtains a polygon with holes that contains the query point *p*, if exists, through *pgn*, and returns *true*. Otherwise, returns *false*.

Oriented_side	*gps.oriented_side(Point_2 q)*	returns either the constant *ON_ORIENTED_BOUNDARY*, *ON_POSITIVE_SIDE*, or *ON_NEGATIVE_SIDE*, iff *p* lies on the boundary, properly on the positive side, or properly on the negative side of *gps* respectively.
Oriented_side	*gps.oriented_side(General_polygon_set_2 other)*	
		returns either the constant *ON_NEGATIVE_SIDE*, *ON_ORIENTED_BOUNDARY*, or *ON_POSITIVE_SIDE*, iff *other* and *gps* are completely disjoint, in contact, or intersect in their interior, respectively.
Oriented_side	*gps.oriented_side(Polygon_2 pgn)*	returns either the constant *ON_NEGATIVE_SIDE*, *ON_ORIENTED_BOUNDARY*, or *ON_POSITIVE_SIDE*, iff *pgn* and *gps* are completely disjoint, in contact, or intersect in their interior, respectively.
Oriented_side	*gps.oriented_side(Polygon_with_holes_2 pgn)*	returns either the constant *ON_NEGATIVE_SIDE*, *ON_ORIENTED_BOUNDARY*, or *ON_POSITIVE_SIDE*, iff *pgn* and *gps* are completely disjoint, in contact, or intersect in their interior, respectively.

Miscellaneous

bool	*gps.is_valid() const*	returns *true* if *gps* represents a valid point set.

See Also

Arrangement_2(page **??**)
ArrangementXMonotoneTraits_2(page 2273)
Nef_2(page **??**)

Concept

GeneralPolygon_2

Refines

GpsTraitsGeneralPolygon_2

Types

GeneralPolygon_2:: X_monotone_curve_2	represents a planar (weakly) x-monotone curve. The type of the geometric mapping of the polygonal edges. It must model the concept ArrTraits::XMonotoneCurve_2.
GeneralPolygon_2:: Curve_iterator	an iterator over the geometric mapping of the polygon edges. Its value type is *X_monotone_curve_2*.
GeneralPolygon_2:: Curve_const_iterator	a const iterator over the geometric mapping of the polygon edges. Its value type is *X_monotone_curve_2*.

Definition

A model of this concept represents a simple general-polygon. The geometric mapping of the edges of the polygon must be x-monotone curves. The concept requires the ability to access these curves. The general polygon represented must be simple. That is, the only points of the plane belonging to two curves are the geometric mapping of the polygon vertices. In addition, the vertices of the represented polygon must be ordered consistently, and the curved must be directed accordingly. Only counterclockwise oriented polygons are valid operands of Boolean set-operations. General polygon that represent holes must be clockwise oriented.

Creation

GeneralPolygon_2	*polygon*;	default constructor.
GeneralPolygon_2	*polygon(other);*	copy constructor.
GeneralPolygon_2	*polygon = other*	assignment operator.

template <class InputIterator>
GeneralPolygon_2 polygon(InputIterator begin, InputIterator end);

constructs a general polygon from a given range of curves.

Access Functions

Curve_iterator	*polygon.curves_begin()*	returns the begin iterator of the curves.
Curve_iterator	*polygon.curves_end()*	returns the past-the-end iterator of the curves.
Curve_const_iterator	*polygon.curves_begin()*	returns the begin const iterator of the curves.
Curve_const_iterator	*polygon.curves_end()*	returns the past-the-end const iterator of the curves.

Modifiers

template <class Iterator>
void *polygon.init(Iterator begin, Iterator end)*

initializes the polygon with the polygonal chain given by the range. The value type of *Iterator* must be *X_monotone_curve_2*.
Precondition: The curves in the range must define a simple polygon.

Has Models

CGAL::General_polygon_2<ArrTraits>

Concept

GeneralPolygonWithHoles_2

Refines

GpsTraitsGeneralPolygonWithHoles_2

Types

GeneralPolygonWithHoles_2:: General_polygon_2	the general-polygon type used to represent the outer boundary and each hole. Must Model the *GeneralPolygon_2* Concept.
GeneralPolygonWithHoles_2:: Hole_const_iterator	a bidirectional iterator over the polygonal holes. Its value type is *General_polygon_2*.

Definition

A model of this concept represents a general polygon with holes. The concept requires the ability to access the general polygon that represents the outer boundary and the general polygons that represent the holes.

Creation

GeneralPolygonWithHoles_2 polygon;	default constructor.
GeneralPolygonWithHoles_2 polygon(other);	copy constructor.
GeneralPolygonWithHoles_2 polygon = other	assignment operator.

GeneralPolygonWithHoles_2 polygon(General_polygon_2 & outer);

constructs a general polygon with holes that has no holes using a given general polygon *outer* as the outer boundary.

template <class InputIterator>
GeneralPolygonWithHoles_2 polygon(General_polygon_2 & outer, InputIterator begin, InputIterator end);

constructs a general polygon with holes using a given general polygon *outer* as the outer boundary and a given range of holes. If *outer* is an empty general polygon, then an unbounded polygon with holes will be created. The holes must be contained inside the outer boundary, and the polygons representing the holes must be simple and pairwise disjoint, except perhaps at the vertices.

Predicates

bool	*polygon.is_unbounded()*	returns *true* if the outer boundary is empty, and *false* otherwise.

Access Functions

General_polygon_2	*polygon.outer_boundary() const*	
		returns the general polygon that represents the outer boundary. Note that this polygon is not necessarily a valid (simple) general polygon because it may be relatively simple.
Hole_const_iterator	*polygon.holes_begin() const*	returns the begin iterator of the holes.
Hole_const_iterator	*polygon.holes_end() const*	returns the past-the-end iterator of the holes.

Has Models

CGAL::General_polygon_with_holes_2<General_polygon>
CGAL::Polygon_with_holes_2<Kernel,Container>
CGAL::Gps_circle_segment_traits_2<Kernel>::Polygon_with_holes_2
CGAL::Gps_traits_2<ArrTraits,GeneralPolygon>::Polygon_with_holes_2

Concept

ArrangementDirectionalXMonotoneTraits_2

Definition

This concept refines the basic arrangement x-monotone traits concept. A model of this concept is able to handle *directed* x-monotone curves that intersect in their interior. Namely, an instance of the *X_monotone_curve_2* type defined by a model of the concept *ArrangementXMonotoneTraits_2* is only required to have a *left* (lexicographically smaller) endpoint and a *right* endpoint. If the traits class is also a model of *ArrangementDirectionalXMonotoneTraits_2*, the x-monotone curve is also required to have a direction, namely one of these two endpoint serves as its *source* and the other as its *target*.

Refines

ArrangementXMonotoneTraits_2

Functor Types

ArrangementDirectionalXMonotoneTraits_2:: Compare_endpoints_xy_2

models the concept *ArrDirectionalTraits::CompareEndpointsXy_2*.

ArrangementDirectionalXMonotoneTraits_2:: Construct_opposite_2

models the concept *ArrDirectionalTraits::ConstructOpposite_2*.

ArrangementDirectionalXMonotoneTraits_2:: Intersect_2

models the concept *ArrDirectionalTraits::Intersect_2*.

ArrangementDirectionalXMonotoneTraits_2:: Split_2

models the concept *ArrDirectionalTraits::Split_2*.

ArrangementDirectionalXMonotoneTraits_2:: Are_mergeable_2

models the concept *ArrDirectionalTraits::AreMergeable_2*.

ArrangementDirectionalXMonotoneTraits_2:: Merge_2

models the concept *ArrDirectionalTraits::Merge_2*.

Creation

ArrangementDirectionalXMonotoneTraits_2	*traits*;	default constructor.
ArrangementDirectionalXMonotoneTraits_2	*traits(other);*	copy constructor
ArrangementDirectionalXMonotoneTraits_2	$traits = other$	assignment operator.

Accessing Functor Objects

Compare_endpoints_xy_2	*traits.compare_endpoints_xy_2_object()*
Construct_opposite_2	*traits.construct_opposite_2_object()*

Has Models

CGAL::Arr_segment_traits_2<Kernel>
CGAL::Arr_non_caching_segment_traits_2<Kernel>
CGAL::Arr_circle_segment_traits_2<Kernel>
CGAL::Arr_conic_traits_2<RatKernel,AlgKernel,NtTraits>
CGAL::Arr_rational_function_traits_2<AlgKernel,NtTraits>
CGAL::Arr_Bezier_curve_traits_2<RatKernel,AlgKernel,NtTraits>
CGAL::Arr_algebraic_segment_traits_2<Coefficient>

See Also

ArrangementXMonotoneTraits_2 (page 2273)

Concept

ArrDirectionalTraits::Intersect_2

Refines

Functor

Has Models

ArrangementDirectionalXMonotoneTraits_2::Intersect_2

Output_iterator *fo.operator()(ArrDirectionalTraits::X_monotone_curve_2 xc1, ArrDirectionalTraits::X_monotone_curve_2 xc2, Output_iterator& oi)*

computes the intersections of *xc1* and *xc2* and inserts them *in an ascending lexicographic xy-order* into the output iterator *oi*. The value-type of *Output_iterator* is *CGAL::Object*, where each *Object* wraps either a *pair<ArrDirectionalTraits::Point_2, ArrDirectionalTraits::Multiplicity>* object, which represents an intersection point with its multiplicity (in case the multiplicity is undefined or unknown, it is set to 0) or an *ArrDirectionalTraits::X_monotone_curve_2* object, representing an overlapping subcurve of *xc1* and *xc2*. In the latter case, the overlapping subcurves are given the direction of *xc1* and *xc2* if their directions are identical. Otherwise, the overlapping subcurves are given an arbitrary direction. The operator returns a past-the-end iterator for the output sequence.

ArrDirectionalTraits::Split_2

Refines

Functor

Has Models

ArrangementDirectionalXMonotoneTraits_2::Split_2

void *fo.operator()(ArrDirectionalTraits::X_monotone_curve_2 xc,*
ArrDirectionalTraits::Point_2 p,
ArrDirectionalTraits::X_monotone_curve_2& xc1,
ArrDirectionalTraits::X_monotone_curve_2& xc2)

accepts an input curve *xc* and a split point *p* in its interior. It splits *xc* at the split point into two subcurves *xc1* and *xc2*, such that *p* is *xc1*'s *right* endpoint and *xc2*'s *left* endpoint. The direction of *xc* is preserved. That is, in case *xc* is directed from left to right, *p* becomes *xc1*'s target and *c2*'s source; otherwise, *p* becomes *xc2*'s target and *xc1*'s source.

Concept

ArrDirectionalTraits::AreMergeable_2

Refines

Functor

Has Models

ArrangementDirectionalXMonotoneTraits_2::Are_mergeable_2

bool *fo.operator()(ArrDirectionalTraits::X_monotone_curve_2 xc1, ArrDirectionalTraits::X_monotone_curve_2 xc2)*

accepts two x-monotone curves *xc1* and *xc2* and determines whether they can be merged to form a single x-monotone curve. *xc1* and *xc2* are mergeable if their underlying curves are identical, they share a common endpoint, and they do not bend to form a non-x-monotone curve.
Precondition: The target point of *xc1* and the source point *xc2* coincide or the source point of *xc2* and the target point *xc2* coincide.

ArrDirectionalTraits::Merge_2

Refines

Functor

Has Models

ArrangementDirectionalXMonotoneTraits_2::Merge_2

void *fo.operator()(ArrDirectionalTraits::X_monotone_curve_2 xc1, ArrDirectionalTraits::X_monotone_curve_2 xc2, ArrDirectionalTraits::X_monotone_curve_2& xc)*

accepts two *mergeable* *x*-monotone curves *xc1* and *xc2* and asigns *xc* with the merged curve. If the target point of *xc1* and the source point of *xc2* coincide; then the source point of *xc1* and the target point of *xc2* become the source and target points of *xc*, respectively. If the target point of *xc2* and the source point of *xc1* coincide; then the source point of *xc2* and the target point of *xc1* become the source and target points of *xc*, respectively.
Precondition: *are_mergeable_2*(*xc1*, *xc2*) is true.

Concept

ArrDirectionalTraits::CompareEndpointsXy_2

Refines

Functor

Has Models

ArrangementDirectionalXMonotoneTraits_2::CompareEndpointsXy_2

Comparison_result *fo(ArrDirectionalTraits::X_monotone_curve_2 xc)*

accepts an input curve *xc* and compares its source and target points. It returns *SMALLER* if the curve is directed from lexicographically left to right, and *LARGER* if it is directed from lexicographically right to left.

ArrDirectionalTraits::ConstructOpposite_2

Concept

Refines

Functor

Has Models

ArrangementDirectionalXMonotoneTraits_2::ConstructOpposite_2

ArrDirectionalTraits::X_monotone_curve_2

fo(ArrDirectionalTraits::X_monotone_curve_2 xc)

accepts an x-monotone curve *xc* and returns its opposite curve, namely a curve whose graph is the same as *xc*'s, and whose source and target are swapped with respect to *xc*'s source and target.

Concept

GeneralPolygonSetTraits_2

Definition

This concept defines the minimal set of geometric predicates needed to perform the Boolean-set operations. It refines the directional x-monotone arrangement-traits concept. In addition to the *Point_2* and *X_monotone_curve_2* types defined in the generalized concept, it defines a type that represents a general polygon and another one that represents general polygon with holes. It also requires operations that operate on these types.

Refines

ArrangementDirectionalXMonotoneTraits_2

Types

GeneralPolygonSetTraits_2:: Polygon_2	represents a general polygon. Must be a model of the *GpsTraitsGeneralPolygon_2* concept.
GeneralPolygonSetTraits_2:: Polygon_with_holes_2	represents a general polygon with holes. Must be a model of the *GpsTraitsGeneralPolygonWithHoles_2* concept.
GeneralPolygonSetTraits_2:: Curve_const_iterator	A const iterator of curves. Its value type is const *X_monotone_curve_2*.

Functor Types

GeneralPolygonSetTraits_2:: Construct_polygon_2

a functor that constructs a general polygon from a range of x-monotone curves. It uses the operator
void operator() (InputIterator begin, Input iterator end, Polygon_2 & pgn),
parameterized by the *InputIterator* type.

GeneralPolygonSetTraits_2:: Construct_general_polygon_with_holes_2

a functor that constructs a general polygon with holes from a general polygon and, optionally, a range of holes. It uses the operator
void operator() (const General_polygon_2& pgn_boundary) or
void operator() (const General_polygon_2& pgn_boundary, HolesInputIterator h_begin, HolesInputIterator h_end) parameterized by the *HolesInputIterator* type.

GeneralPolygonSetTraits_2:: Construct_outer_boundary

A functor for constructing the outer boundary of a polygon with holes. It uses the operator
General_polygon_2 operator()(const General_polygon_with_holes_2& pol_wh).

GeneralPolygonSetTraits_2:: Construct_holes	A functor for constructing the container of holes of a polygon with holes. It returns the begin/end iterators for the holes It uses the operator *std::pair<Hole_const_iterator, Hole_const_iterator> operator()(const General_polygon_with_holes_2& pol_wh).*
GeneralPolygonSetTraits_2:: Is_unbounded	A functor for checking if polygon with holes has no outer boundary. It uses the operator *bool operator()(const General_polygon_with_holes_2& pol_wh).*

Creation

GeneralPolygonSetTraits_2 traits; — default constructor.

GeneralPolygonSetTraits_2 traits(other); — copy constructor

GeneralPolygonSetTraits_2

traits = other — assignment operator.

Accessing Functor Objects

Construct_polygon_2 *traits.construct_polygon_2_object()*

returns a functor that constructs a polygon.

Construct_curves_2 *traits.construct_curves_2_object()*

returns a functor that obtains the curves of a polygon.

Construct_general_polygon_with_holes_2

traits.construct_polygon_with_holes_2_object() const

returns a functor that constructs a polygon with holes.

Construct_outer_boundary

traits.construct_outer_boundary_object() const

returns a functor that obtains the outer boundary of a polygon with holes.

Construct_holes *traits.construct_holes_object() const*

returns a functor that obtains begin/end iterators over a container of holes.

Is_unbounded	*traits.construct_is_unbounded_object()*	
		returns a functor that determines if the polygon with holes is unbounded

Has Models

CGAL::Gps_segment_traits_2<Kernel,Container,ArrSegmentTraits>
CGAL::Gps_circle_segment_traits_2<Kernel>
CGAL::Gps_traits_2<ArrTraits,GeneralPolygon>

See Also

ArrangementDirectionalXMonotoneTraits_2(page 1522)

CGAL::General_polygon_2<ArrTraits>

Class

Definition

The class *General_polygon_2<ArrTraits>* models the concept *GeneralPolygon_2*. It represents a simple general-polygon. It is parameterized with the type *ArrTraits* that models the concept *ArrangementDirectionalXMonotoneTraits*. The latter is a refinement of the concept *ArrangementXMonotoneTraits_2*. In addition to the requirements of the concept *ArrangementXMonotoneTraits_2*, a model of the concept *ArrangementDirectionalXMonotoneTraits* must support the following functions:

- Given an x-monotone curve, construct its opposite curve.
- Given an x-monotone curve, compare its two endpoints lexicographically.

This class supports a few convenient operations in addition to the requirements that the concept *GeneralPolygon_2* lists.

#include <CGAL/General_polygon_2.h>

Types

General_polygon_2<ArrTraits>:: Size	number of edges size type.

Creation

Operations

Size	*polygon.size()*	returns the number of edges of *polygon*.

Modifiers

void	*polygon.clear()*	clears *polygon*.
void	*polygon.reverse_orientation()*	reverses the orientation of the polygon. *Precondition*: *is_simple()*.

Predicates

bool	*polygon.is_empty()*	returns *true* if *polygon* is empty, and *false* otherwise.
Orientation	*polygon.orientation()*	returns the orientation of *polygon*. *Precondition*: *is_simple()*.

Is Model for the Concepts

GeneralPolygon_2

Class

CGAL::Polygon_with_holes_2<Kernel,Container>

Definition

The class *Polygon_with_holes_2<Kernel,Container>* models the concept *GeneralPolygonWithHoles_2*. It represents a linear polygon with holes. It is parameterized with two types (*Kernel* and *Container*) that are used to instantiate the type *CGAL::Polygon_2<Kernel,Container>*. The latter is used to represents the outer boundary and the boundary of the holes (if any exist).

#include <CGAL/Polygon_with_holes_2.h>

Is Model for the Concepts

GeneralPolygonWithHoles_2

CGAL::Polygon_set_2<Kernel,Container,Dcel>

Definition

The class *Polygon_set_2<Kernel,Container,Dcel>* represents sets of linear polygons with holes. The first two template parameters (*Kernel* and *Container*) are used to instantiate the type *CGAL::Polygon_2< Kernel,Container>*. This type is used to represents the outer boundary of every set member and the boundaries of all holes of every set member.

The third template parameter *Dcel* should be instantiated with a model of the concept *GeneralPolygonSetDcel*. It is instantiated by default with the type *Gps_default_dcel<Traits>*. You can override this default, with a different DCEL class, typically an extension of the *Gps_default_dcel* class template. Overriding the default is necessary only if you intend to obtain the underlying internal arrangement and process it further.

#include <CGAL/Polygon_set_2.h>

Inherits From

General_polygon_set_2<Gps_segment_traits_2<Kernel,Container> >

See Also

General_polygon_set_2(page 1511)
Gps_segment_traits_2(page 1537)

Class

CGAL::General_polygon_with_holes_2<Polygon>

Definition

The class *General_polygon_with_holes_2<Polygon>* models the concept *GeneralPolygonWithHoles_2*. It represents a general polygon with holes. It is parameterized with a type *Polygon* used to define the exposed type *General_polygon_2*. This type represents the outer boundary of the general polygon and the outer boundaries of each hole.

#include <CGAL/General_polygon_with_holes_2.h>

typedef Polygon	*General_polygon_2;*

Is Model for the Concepts

GeneralPolygonWithHoles_2

CGAL::Gps_segment_traits_2<Kernel,Container,ArrSegmentTraits>

Class

Definition

The traits class *Gps_segment_traits_2<Kernel,Container,ArrSegmentTraits>* models the concept *GeneralPolygonSetTraits_2*. It enables Boolean set-operations on (linear) polygons. It defines the exposed type *Polygon_2* to be *CGAL::Polygon_2<Kernel,Container>*. By default, the template parameter *Container* is instantiated by *std::vector<Kernel::Point_2>* and the template parameter *ArrSegmentTraits* is instantiated by *CGAL::Arr_segment_traits_2<Kernel>*.

#include <CGAL/Gps_segment_traits_2.h>

typedef CGAL::Polygon_2<Kernel,Container>

Polygon_2;

Is Model for the Concepts

GeneralPolygonSetTraits_2

See Also

CGAL::Arr_segment_traits_2<Kernel>(page **??**)

Class

CGAL::Gps_circle_segment_traits_2<Kernel>

Definition

The traits class *Gps_circle_segment_traits_2<Kernel>* models the *GeneralPolygonSetTraits_2* concept. It enables Boolean set-operations on general polygons bounded by linear segments or circular arcs. It should be parameterized with a kernel.

#include <CGAL/Gps_circle_segment_traits_2.h>

Is Model for the Concepts

GeneralPolygonSetTraits_2

See Also

CGAL::Arr_circle_segment_traits_2<Kernel>(page **??**)

CGAL::Gps_traits_2<ArrTraits,GeneralPolygon_t>

Class

Definition

The traits class *Gps_traits_2<ArrTraits,GeneralPolygon_t>* models the concept *GeneralPolygonSetTraits_2*. It inherits from the instantiated type of the template parameter *ArrTraits*, which must model the concept *ArrangementDirectionalXMonotoneTraits*, (which in turn refines the concept *ArrangementXMonotoneTraits*). The template parameter *GeneralPolygon_t* must be instantiated with a model of the concept of *GpsTraitsGeneralPolygon_2*. By default, the latter is instantiated by *CGAL::General_polygon_2<ArrTraits>* .

#include <CGAL/Gps_traits_2.h>

Is Model for the Concepts

GeneralPolygonSetTraits_2

Class

CGAL::Gps_default_dcel<Traits>

Definition

The class *Gps_default_dcel<Traits>* is the default DCEL class used by the *General_polygon_set_2* and *Polygon_set_2* class-templates to represent the underlying internal *Arrangement_2* data structure.

#include <CGAL/Boolean_set_operations_2/Gps_default_dcel.h>

Is Model for the Concepts

GeneralPolygonSetDcel

Inherits From

Arr_dcel_base<Arr_vertex_base<typename Traits_::Point_2>,
Arr_halfedge_base<typename Traits_::X_monotone_curve_2>,
Gps_face_base>

Types

Gps_default_dcel<Traits>:: template <class T> rebind

allows the rebinding of the DCEL with a different traits class T.

GeneralPolygonSetDcel

Concept

Definition

A doubly-connected edge-list (DCEL for short) data-structure. It consists of three containers of records: vertices V, halfedges E, and faces F. It maintains the incidence relation among them. The halfedges are ordered in pairs sometimes referred to as twins, such that each halfedge pair represent an edge.

A model of the GeneralPolygonSetDcel concept must provide the following types and operations. (In addition to the requirements here, the local types *Vertex*,*Halfedge*, *Face Hole* and *Isolated_vertex* must be models of the concepts *ArrangementDcelVertex*(page 2236), *ArrangementDcelHalfedge*(page 2238), *GeneralPolygonSetDcelFace* (page 1544), *ArrangementDcelHole*(page 2243), and *ArrangementDcelIsolatedVertex* (page 2244) respectively.) Notice that this concept differs from the concept *ArrangemenDcel* only in the type *Face*.

Types

GeneralPolygonSetDcel:: Vertex	the vertex type.
GeneralPolygonSetDcel:: Halfedge	the halfedge type.
GeneralPolygonSetDcel:: Face	the face type.
GeneralPolygonSetDcel:: Hole	the hole type.
GeneralPolygonSetDcel:: Isolated_vertex	the isolated vertex type.
GeneralPolygonSetDcel:: Size	used to represent size values (e.g., *size_t*).
GeneralPolygonSetDcel:: Vertex_iterator	a bidirectional iterator over the vertices. Its value-type is *Vertex*.
GeneralPolygonSetDcel:: Halfedge_iterator	a bidirectional iterator over the halfedges. Its value-type is *Halfedge*.
GeneralPolygonSetDcel:: Face_iterator	a bidirectional iterator over the faces. Its value-type is *Face*.

The non-mutable iterators *Vertex_const_iterator*, *Halfedge_const_iterator* and *Face_const_iterator* are also defined.

Creation

GeneralPolygonSetDcel dcel; — constructs an empty DCEL with one unbounded face.

*Face** *dcel.assign(Self other, const Face *uf)*

assigns the contents of the *other* DCEL whose unbounded face is given by *uf*, to *dcel*. The function returns a pointer to the unbounded face of *dcel* after the assignment.

Access Functions

Size *dcel.size_of_vertices() const*

returns the number of vertices.

Size *dcel.size_of_halfedges() const*

returns the number of halfedges (always even).

Size *dcel.size_of_faces() const*

returns the number of faces.

Size *dcel.size_of_holes() const*

returns the number of holes (the number of connected components).

Size *dcel.size_of_isolated_vertices() const*

returns the number of isolated vertices.

The following operations have an equivalent *const* operations that return the corresponding non-mutable iterators:

Vertex_iterator *dcel.vertices_begin()* returns a begin-iterator of the vertices in *dcel*.
Vertex_iterator *dcel.vertices_end()* returns a past-the-end iterator of the vertices in *dcel*.

Halfedge_iterator *dcel.halfedges_begin()*

returns a begin-iterator of the halfedges in *dcel*.

Halfedge_iterator *dcel.halfedges_end()* returns a past-the-end iterator of the halfedges in *dcel*.

Vertex_iterator *dcel.faces_begin()* returns a begin-iterator of the faces in *dcel*.
Vertex_iterator *dcel.faces_end()* returns a past-the-end iterator of the faces in *dcel*.

Modifiers

The following operations allocate a new element of the respective type. Halfedges are always allocated in pairs of opposite halfedges. The halfedges and their opposite pointers are automatically set.

*Vertex** *dcel.new_vertex()* creates a new vertex.
*Halfedge** *dcel.new_edge()* creates a new pair of twin halfedges.
*Face** *dcel.new_face()* creates a new face.
*Hole** *dcel.new_hole()* creates a new hole record.
*Isolated_vertex** *dcel.new_isolated_vertex()*

creates a new isolated vertex record.

void *dcel.delete_vertex(Vertex* v)*

deletes the vertex *v*.

void	*dcel.delete_edge(Halfedge* e)*	
		deletes the halfedge *e* as well as its twin.
void	*dcel.delete_face(Face* f)*	
		deletes the face *f*.
void	*dcel.delete_hole(Hole* ho)*	
		deletes the hole *ho*.
void	*dcel.delete_isolated_vertex(Isolated_vertex* iv)*	
		deletes the isolated vertex *iv*.

Has Models

Arr_dcel_base<V,H,F>(page 2245)
Arr_default_dcel<Traits> (page 2247)
Arr_face_extended_dcel<Traits,FData,V,H,F> (page 2248)
Arr_extended_dcel<Traits,VData,HData,FData,V,H,F> (page 2249)

See Also

ArrangementDcelVertex(page 2236)
ArrangementDcelHalfedge(page 2238)
GeneralPolygonSetDcelFace(page 1544)
ArrangementDcelHole(page 2243)
ArrangementDcelIsolatedVertex (page 2244)

Concept

GeneralPolygonSetDcelFace

Definition

A face record in a DCEL data structure used by the *General_polygon_set_2* and *Polygon_set_2* class-templates to represent the underlying internal *Arrangement_2* data structure.

Refines

ArrangementDcelFace

Creation

GeneralPolygonSetDcelFace f;		default constructor.
void	*f.assign(Self other)*	assigns *f* with the contents of the *other* face.

Access Functions

bool	*f.contained() const*	returns *true* if the face is contained in the general-polygon set, and *false* otherwise.
bool	*f.visited()*	returns *true* if the face has been visited, and *false* otherwise. This is used internally by the some of the operations of the *General_polygon_set_2* class that traverse the arrangement faces.

Modifiers

void	*f.set_contained(bool flag)*	marks the face as contained (if *flag* is *true*), or as a hole (if it is *false*).
void	*f.set_visited(bool flag)*	marks the face as visited (if *flag* is *true*), or as not visited (if it is *false*). This is used internally by the some of the operations of the *General_polygon_set_2* class that traverse the arrangement faces.

See Also

ArrangementDcel(page 2232)
ArrangementDcelVertex(page 2236)
ArrangementDcelHalfedge(page 2238)

CGAL::complement

Definition

#include <CGAL/Boolean_set_operations_2.h>

void *complement(Type pgn, Type & res)*

Each one of these functions computes the complement of a given polygon *pgn*, and stores the resulting polygon with holes in *res*.

Arg Type
Polygon_2
General_polygon_2

OutputIterator *complement(Type pgn, OutputIterator oi)*

Each one of these functions computes the complement of a given polygon *pgn*, inserts the resulting polygons with holes into an output container through a given output iterator *oi*, and returns the output iterator. The value type of the *OutputIterator* is either *Polygon_with_holes_2* or *General_polygon_with_holes_2*.

Arg Type
Polygon_with_holes_2
General_polygon_with_holes_2

```
template <class Kernel, class Container>
void            complement( Polygon_2<Kernel, Container> pgn,
                            Polygon_with_holes_2<Kernel, Container> & res)

template <class Traits>
void            complement( General_polygon_2<Traits> pgn,
                            General_polygon_with_holes_2<Traits> & res)


template <class Traits, class OutputIterator>
OutputIterator  complement( Polygon_with_holes_2<Kernel, Container> pgn, OutputIterator oi)
template <class Traits, class OutputIterator>
OutputIterator  complement( General_polygon_with_holes_2<General_polygon_2<Traits> > pgn,
                            OutputIterator oi)
```

See Also

CGAL::do_intersect . . . . page 2218
CGAL::intersection . . . . page 1553
CGAL::join . . . . page 1555

CGAL::difference

Definition

```
#include <CGAL/Boolean_set_operations_2.h>
```

```
OutputIterator    difference( Type1 p1, Type2 p2, OutputIterator oi)
```

Each one of these functions computes the difference between two given polygons *p1* and *p2*, and inserts the resulting polygons with holes into an output container through the output iterator *oi*. The value type of the *OutputIterator* is either *Polygon_with_holes_2* or *General_polygon_with_holes_2*.

Arg 1 Type	Arg 2 Type
Polygon_2	*Polygon_2*
Polygon_2	*Polygon_with_holes_2*
Polygon_with_holes_2	*Polygon_2*
Polygon_with_holes_2	*Polygon_with_holes_2*
General_polygon_2	*General_polygon_2*
General_polygon_2	*General_polygon_with_holes_2*
General_polygon_with_holes_2	*General_polygon_2*
General_polygon_with_holes_2	*General_polygon_with_holes_2*

```
template <class Kernel, class Container, class OutputIterator>
OutputIterator    difference( Polygon_2<Kernel, Container> p1,
                              Polygon_2<Kernel, Container> p2,
                              OutputIterator oi)
```

```
template <class Kernel, class Container, class OutputIterator>
OutputIterator    difference( Polygon_2<Kernel, Container> p1,
                              Polygon_with_holes_2<Kernel,Container> p2,
                              OutputIterator oi)
```

```
template <class Kernel, class Container, class OutputIterator>
OutputIterator    difference( Polygon_with_holes_2<Kernel, Container> p1,
                              Polygon_2<Kernel, Container> p2,
                              OutputIterator oi)
```

```
template <class Kernel, class Container, class OutputIterator>
OutputIterator    difference( Polygon_with_holes_2<Kernel, Container> p1,
                              Polygon_with_holes_2<Kernel, Container> p2,
                              OutputIterator oi)
```

```
template <class Traits, class OutputIterator>
OutputIterator    difference( General_polygon_2<Traits> p1,
                              General_polygon_2<Traits> p2,
                              OutputIterator oi)
```

```
template <class Traits, class OutputIterator>
OutputIterator    difference( General_polygon_with_holes_2<General_polygon_2<Traits> > p1,
                              General_polygon_2<Traits> p2,
                              OutputIterator oi)
```

template <class Traits, class OutputIterator>
OutputIterator difference(General_polygon_2<Traits> p1,
General_polygon_with_holes_2<General_polygon_2<Traits> > p2,
OutputIterator oi)

template <class Polygon, class OutputIterator>
OutputIterator difference(General_polygon_with_holes_2<Polygon> p1,
General_polygon_with_holes_2<Polygon> p2)

See Also

CGAL::do_intersect . . . . page 2218
CGAL::intersection . . . . page 1553
CGAL::join . . . . page 1555
CGAL::symmetric_difference . . . . page 1557

CGAL::do_intersect

Definition

#include <CGAL/Boolean_set_operations_2.h>

bool *do_intersect(Type1 p1, Type2 p2)*

Each one of these functions return *true*, if the two given polygons *p1* and *p2* intersect in their interior, and *false* otherwise.

Arg 1 Type	**Arg 2 Type**
Polygon_2	*Polygon_2*
Polygon_2	*Polygon_with_holes_2*
Polygon_with_holes_2	*Polygon_2*
Polygon_with_holes_2	*Polygon_with_holes_2*
General_polygon_2	*General_polygon_2*
General_polygon_2	*General_polygon_with_holes_2*
General_polygon_with_holes_2	*General_polygon_2*
General_polygon_with_holes_2	*General_polygon_with_holes_2*

template <class Kernel, class Container>
bool *do_intersect(Polygon_2<Kernel, Container> p1, Polygon_2<Kernel, Container> p2)*
template <class Kernel, class Container>
bool *do_intersect(Polygon_2<Kernel, Container> p1, Polygon_with_holes_2<Kernel, Container> p2)*
template <class Kernel, class Container>
bool *do_intersect(Polygon_with_holes_2<Kernel, Container> p1, Polygon_2<Kernel, Container> p2)*
template <class Kernel, class Container>
bool *do_intersect(Polygon_with_holes_2<Kernel, Container> p1,*
Polygon_with_holes_2<Kernel, Container> p2)

template <class Traits>
bool *do_intersect(General_polygon_2<Traits> p1, General_polygon_2<Traits> p2)*
template <class Traits>
bool *do_intersect(General_polygon_2<Traits> p1,*
General_polygon_with_holes_2<General_polygon_2<Traits> > p2)

template <class Traits>
bool *do_intersect(General_polygon_with_holes_2<General_polygon_2<Traits> > p1,*
General_polygon_2<Traits> p2)

template <class Polygon>
bool *do_intersect(General_polygon_with_holes_2<Polygon> p1,*
General_polygon_with_holes_2<Polygon> p2)

template <class InputIterator>
bool *do_intersect(InputIterator begin, InputIterator end)*

Returns *true*, if the set of general polygons (or general polygons with holes) in the given range intersect in their interior, and *false* otherwise. (The value type of the input iterator is used to distinguish between the two).

template <class InputIterator1, class InputIterator2>
bool do_intersect(InputIterator1 pgn_begin1,
InputIterator1 pgn_end1,
InputIterator2 pgn_begin2,
InputIterator2 pgn_end2)

Returns *true*, if the set of general polygons and general polygons with holes in the given two ranges respectively intersect in their interior, and *false* otherwise.

See Also

CGAL::intersection . . . . page 1553
CGAL::join . . . . page 1555
CGAL::difference . . . . page 1547
CGAL::symmetric_difference . . . . page 1557

CGAL::oriented_side

Definition

```
#include <CGAL/Boolean_set_operations_2.h>
```

```
Oriented_side    oriented_side( Type1 p1, Type2 p2)
```

Each one of these functions returns *ON_POSITIVE_SIDE* if the two given polygons *p1* and *p2* intersect in their interior, *ON_NEGATIVE_SIDE* if *p1* and *p2* do not intersect at all, and *ON_ORIENTED_BOUNDARY* if *p1* and *p2* intersect only in their boundaries.

Arg 1 Type	**Arg 2 Type**
Polygon_2	*Polygon_2*
Polygon_2	*Polygon_with_holes_2*
Polygon_with_holes_2	*Polygon_2*
Polygon_with_holes_2	*Polygon_with_holes_2*
General_polygon_2	*General_polygon_2*
General_polygon_2	*General_polygon_with_holes_2*
General_polygon_with_holes_2	*General_polygon_2*
General_polygon_with_holes_2	*General_polygon_with_holes_2*

```
template <class Kernel, class Container>
Oriented_side    oriented_side( Polygon_2<Kernel, Container> p1, Polygon_2<Kernel, Container> p2)
template <class Kernel, class Container>
Oriented_side    oriented_side( Polygon_2<Kernel, Container> p1,
                                Polygon_with_holes_2<Kernel, Container> p2)

template <class Kernel, class Container>
Oriented_side    oriented_side( Polygon_with_holes_2<Kernel, Container> p1,
                                Polygon_2<Kernel, Container> p2)

template <class Kernel, class Container>
Oriented_side    oriented_side( Polygon_with_holes_2<Kernel, Container> p1,
                                Polygon_with_holes_2<Kernel, Container> p2)

template <class Traits>
Oriented_side    oriented_side( General_polygon_2<Traits> p1, General_polygon_2<Traits> p2)
template <class Traits>
Oriented_side    oriented_side( General_polygon_2<Traits> p1,
                                General_polygon_with_holes_2<General_polygon_2<Traits> > p2)

template <class Traits>
Oriented_side    oriented_side( General_polygon_with_holes_2<General_polygon_2<Traits> > p1,
                                General_polygon_2<Traits> p2)

template <class Polygon>
Oriented_side    oriented_side( General_polygon_with_holes_2<Polygon> p1,
                                General_polygon_with_holes_2<Polygon> p2)
```

See Also

CGAL::do_intersect .... page 2218

CGAL::intersection

Definition

#include <CGAL/Boolean_set_operations_2.h>

OutputIterator intersection(Type1 p1, Type2 p2, OutputIterator oi)

Each one of these functions computes the intersection of two given polygons *p1* and *p2*, inserts the resulting polygons with holes into an output container through a given output iterator *oi*, and returns the output iterator. The value type of the *OutputIterator* is either *Polygon_with_holes_2* or *General_polygon_with_holes_2*.

Arg 1 Type	**Arg 2 Type**
Polygon_2	*Polygon_2*
Polygon_2	*Polygon_with_holes_2*
Polygon_with_holes_2	*Polygon_2*
Polygon_with_holes_2	*Polygon_with_holes_2*
General_polygon_2	*General_polygon_2*
General_polygon_2	*General_polygon_with_holes_2*
General_polygon_with_holes_2	*General_polygon_2*
General_polygon_with_holes_2	*General_polygon_with_holes_2*

```
template <class Kernel, class Container, class OutputIterator>
OutputIterator  intersection( Polygon_2<Kernel, Container> p1,
                              Polygon_2<Kernel, Container> p2,
                              OutputIterator oi)
```

```
template <class Kernel, class Container, class OutputIterator>
OutputIterator  intersection( Polygon_2<Kernel, Container> p1,
                              Polygon_with_holes_2<Kernel, Container> p2,
                              OutputIterator oi)
```

```
template <class Kernel, class Container, class OutputIterator>
OutputIterator  intersection( Polygon_with_holes_2<Kernel, Container> p1,
                              Polygon_2<Kernel, Container> p2,
                              OutputIterator oi)
```

```
template <class Kernel, class Container, class OutputIterator>
OutputIterator  intersection( Polygon_with_holes_2<Kernel, Container> p1,
                              Polygon_with_holes_2<Kernel, Container> p2,
                              OutputIterator oi)
```

```
template <class Traits, class OutputIterator>
OutputIterator  intersection( General_polygon_2<Traits> p1,
                              General_polygon_2<Traits> p2,
                              OutputIterator oi)
```

```
template <class Traits, class OutputIterator>
OutputIterator  intersection( General_polygon_with_holes_2<General_polygon_2<Traits> > p1,
                              General_polygon_2<Traits> p2,
                              OutputIterator oi)
```

template <class Traits, class OutputIterator>
OutputIterator intersection(General_polygon_2<Traits> p1,
General_polygon_with_holes_2<General_polygon_2<Traits> > p2,
OutputIterator oi)

template <class Polygon, class OutputIterator>
OutputIterator intersection(General_polygon_with_holes_2<Polygon> p1,
General_polygon_with_holes_2<Polygon> p2,
OutputIterator oi)

template <class InputIterator, class OutputIterator>
OutputIterator intersection(InputIterator begin, InputIterator end, OutputIterator oi)

Computes the intersection of the general polygons (or general polygons with holes) in the given range. (The value type of the input iterator is used to distinguish between the two.) The result, represented by a set of general polygon with holes, is inserted into an output container through a given output iterator *oi*. The output iterator is returned. The value type of the *OutputIterator* is *Traits::Polygon_with_holes_2*.

template <class InputIterator1, class InputIterator2, class OutputIterator>
OutputIterator intersection(InputIterator1 pgn_begin1,
InputIterator1 pgn_end1,
InputIterator2 pgn_begin2,
InputIterator2 pgn_end2,
OutputIterator oi)

Computes the intersection of the general polygons and general polygons with holes in the given two ranges. The result, represented by a set of general polygon with holes, is inserted into an output container through a given output iterator *oi*. The output iterator is returned. The value type of the *OutputIterator* is *Traits::Polygon_with_holes_2*.

See Also

CGAL::do_intersect . . . . page 2218
CGAL::join . . . . page 1555
CGAL::difference . . . . page 1547
CGAL::symmetric_difference . . . . page 1557

CGAL::join

Function

Definition

#include <CGAL/Boolean_set_operations_2.h>

bool *join(Type1 p1, Type2 p2, General_polygon_with_holes_2 & p)*

Each one of these functions computes the union of two given polygons *p1* and *p2*. If the two given polygons overlap, it returns *true*, and places the resulting polygon in *p*. Otherwise, it returns *false*.

Arg 1 Type	Arg 2 Type
Polygon_2	*Polygon_2*
Polygon_2	*Polygon_with_holes_2*
Polygon_with_holes_2	*Polygon_2*
Polygon_with_holes_2	*Polygon_with_holes_2*
General_polygon_2	*General_polygon_2*
General_polygon_2	*General_polygon_with_holes_2*
General_polygon_with_holes_2	*General_polygon_2*
General_polygon_with_holes_2	*General_polygon_with_holes_2*

```
template <class Kernel, class Container>
bool        join( Polygon_2<Kernel, Container> p1,
                  Polygon_2<Kernel, Container> p2,
                  General_polygon_with_holes_2<Polygon_2<Kernel, Container> > & p)

template <class Kernel, class Container>
bool        join( Polygon_2<Kernel, Container> p1,
                  Polygon_with_holes_2<Kernel,Container> p2,
                  General_polygon_with_holes_2<Polygon_2<Kernel, Container> > & p)

template <class Kernel, class Container>
bool        join( Polygon_with_holes_2<Kernel, Container> p2,
                  Polygon_2<Kernel, Container> p1,
                  General_polygon_with_holes_2<Polygon_2<Kernel, Container> > & p)

template <class Kernel, class Container>
bool        join( Polygon_with_holes_2<Kernel, Container> p2,
                  Polygon_with_holes_2<Kernel, Container> p1,
                  General_polygon_with_holes_2<Polygon_2<Kernel, Container> > & p)

template <class Traits>
bool        join( General_polygon_2<Traits> p1,
                  General_polygon_2<Traits> p2,
                  General_polygon_with_holes_2<General_polygon_2<Traits> > & p)

template <class Traits>
bool        join( General_polygon_2<Traits> p1,
                  General_polygon_with_holes_2<General_polygon_2<Traits> > p2,
                  General_polygon_with_holes_2<General_polygon_2<Traits> > & p)
```

template <class Traits>
bool *join(General_polygon_with_holes_2<General_polygon_2<Traits> > p2, General_polygon_2<Traits> p1, General_polygon_with_holes_2<General_polygon_2<Traits> > & p)*

template <class Polygon>
bool *join(General_polygon_with_holes_2<Polygon> p1, General_polygon_with_holes_2<Polygon> p2, Traits::Polygon_with_holes_2 & p)*

template <class InputIterator, class OutputIterator>
OutputIterator *join(InputIterator begin, InputIterator end, OutputIterator oi)*

Computes the union of the general polygons (or general polygons with holes) in the given range. (The value type of the input iterator is used to distinguish between the two.) The result, represented by a set of general polygon with holes, is inserted into an output container through a given output iterator *oi*. The output iterator is returned. The value type of the *OutputIterator* is *Traits::Polygon_with_holes_2*.

template <class InputIterator1, class InputIterator2, class OutputIterator>
OutputIterator *join(InputIterator1 pgn_begin1, InputIterator1 pgn_end1, InputIterator2 pgn_begin2, InputIterator2 pgn_end2, OutputIterator oi)*

Computes the union of the general polygons and general polygons with holes in the given two ranges. The result, represented by a set of general polygon with holes, is inserted into an output container through a given output iterator *oi*. The output iterator is returned. The value type of the *OutputIterator* is *Traits::Polygon_with_holes_2*.

See Also

CGAL::do_intersect . . . . page 2218
CGAL::intersection . . . . page 1553
CGAL::difference . . . . page 1547
CGAL::symmetric_difference . . . . page 1557

CGAL::symmetric_difference

Definition

#include <CGAL/Boolean_set_operations_2.h>

OutputIterator intersection(Type1 p1, Type2 p2, OutputIterator oi)

Each one of these functions computes the symmetric difference between two given polygons *p1* and *p2*, and inserts the resulting polygons with holes into an output container through the output iterator *oi*. The value type of the *OutputIterator* is either *Polygon_with_holes_2* or *General_polygon_with_holes_2*.

Arg 1 Type	Arg 2 Type
Polygon_2	*Polygon_2*
Polygon_2	*Polygon_with_holes_2*
Polygon_with_holes_2	*Polygon_2*
Polygon_with_holes_2	*Polygon_with_holes_2*
General_polygon_2	*General_polygon_2*
General_polygon_2	*General_polygon_with_holes_2*
General_polygon_with_holes_2	*General_polygon_2*
General_polygon_with_holes_2	*General_polygon_with_holes_2*

template <class Kernel, class Container, class OutputIterator>
OutputIterator symmetric_difference(Polygon_2<Kernel, Container> p1,
Polygon_2<Kernel, Container> p2,
OutputIterator oi)

template <class Kernel, class Container, class OutputIterator>
OutputIterator symmetric_difference(Polygon_2<Kernel, Container> p1,
Polygon_with_holes_2<Kernel, Container> p2,
OutputIterator oi)

template <class Kernel, class Container, class OutputIterator>
OutputIterator symmetric_difference(Polygon_with_holes_2<Kernel, Container> p1,
Polygon_2<Kernel, Container> p2,
OutputIterator oi)

template <class Kernel, class Container, class OutputIterator>
OutputIterator symmetric_difference(Polygon_with_holes_2<Kernel, Container> p1,
Polygon_with_holes_2<Kernel, Container> p2,
OutputIterator oi)

template <class Traits, class OutputIterator>
OutputIterator symmetric_difference(General_polygon_2<Traits> p1,
General_polygon_2<Traits> p2,
OutputIterator oi)

template <class Traits, class OutputIterator>
OutputIterator symmetric_difference(General_polygon_with_holes_2<General_polygon_2<Traits> > p1,
General_polygon_2<Traits> p2,
OutputIterator oi)

template <class Traits, class OutputIterator>
OutputIterator *symmetric_difference(General_polygon_2<Traits> p1,*
General_polygon_with_holes_2<General_polygon_2<Traits> > p2,
OutputIterator oi)

template <class Polygon, class OutputIterator>
OutputIterator *symmetric_difference(General_polygon_with_holes_2<Polygon> p1,*
General_polygon_with_holes_2<Polygon> p2,
OutputIterator oi)

template <class InputIterator, class OutputIterator>
OutputIterator *symmetric_difference(InputIterator begin, InputIterator end, OutputIterator oi)*

Computes the symmetric difference of the general polygons (or general polygons with holes) in the given range. A point is contained in the symmetric difference, if and only if it is contained in odd number of input polygons. (The value type of the input iterator is used to distinguish between the two.) The result, represented by a set of general polygon with holes, is inserted into an output container through a given output iterator *oi*. The output iterator is returned. The value type of the *OutputIterator* is *Traits::Polygon_with_holes_2*.

template <class InputIterator1, class InputIterator2, class OutputIterator>
OutputIterator *symmetric_difference(InputIterator1 pgn_begin1,*
InputIterator1 pgn_end1,
InputIterator2 pgn_begin2,
InputIterator2 pgn_end2,
OutputIterator oi)

Computes the union of the general polygons and general polygons with holes in the given two ranges. A point is contained in the symmetric difference, if and only if it is contained in odd number of input polygons. The result, represented by a set of general polygon with holes, is inserted into an output container through a given output iterator *oi*. The output iterator is returned. The value type of the *OutputIterator* is *Traits::Polygon_with_holes_2*.

See Also

CGAL::do_intersect . . . . page 2218
CGAL::intersection . . . . page 1553
CGAL::join . . . . page 1555
CGAL::difference . . . . page 1547

CGAL::connect_holes

Function

Definition

#include <CGAL/connect_holes.h>

template <class Kernel, class Container, class OutputIterator>
OutputIterator *connect_holes(Polygon_with_holes_2<Kernel,Container> pwh, OutputIterator oi)*

Connects the holes of *pwh* with its outer boundary. This is done by locating the topmost vertex in each hole in the polygon with holes *pwh*, and connecting it by a vertical segment to the polygon feature located directly above it (a vertex or an edge of the outer boundary, or of another hole). The function produces an output sequence of points, which corresponds to the traversal of the vertices of the input polygon; this traversal starts from the outer boundary and moves to the holes using the auxiliary vertical segments that were added to connect the polygon with its holes. The value-type of *oi* is *Kernel::Point_2*.

Precondition: The input polygon with holes *pwh* is bounded (namely it has a valid outer boundary).

Function

CGAL::operator<<

Definition

This operator exports a polygon with holes, a general polygon, or a general polygon with holes *P* to the output stream *out*. The output is in ASCII format.

An ASCII and a binary format exist. The format can be selected with the CGAL modifiers for streams, *set_ascii_mode* and *set_binary_mode* respectively. The modifier *set_pretty_mode* can be used to allow for (a few) structuring comments in the output. Otherwise, the output would be free of comments. The default for writing is ASCII without comments.

#include <CGAL/Polygon_with_holes_2.h> *template <class Kernel, class Container>*

ostream& *ostream& out << CGAL::Polygon_with_holes_2<Kernel,Container> P*

The number of points of the outer boundary is exported followed by the points themselves in counterclockwise order. Then, the number of holes is exported, and for each hole, the number of points on its outer boundary is exported followed by the points themselves in clockwise order.

#include <CGAL/General_polygon_2.h> *template <class ArrTraits>*

ostream& *ostream& out << CGAL::General_polygon_2<ArrTraits> P*

The number of curves of the outer boundary is exported followed by the curves themselves in counterclockwise order.

#include <CGAL/General_polygon_with_holes_2.h> *template <class Polygon>*

ostream& *ostream& out << CGAL::General_polygon_with_holes_2<Polygon> P*

The number of curves of the outer boundary is exported followed by the curves themselves in counterclockwise order. Then, the number of holes is exported, and for each hole, the number of curves on its outer boundary is exported followed by the curves themselves in clockwise order.

See Also

CGAL::Polygon_2<PolygonTraits_2, Container> . . . . page 1472
CGAL::General_polygon_2<ArrTraits> . . . . page 1533
CGAL::General_polygon_with_holes_2<Polygon> . . . . page 1536
operator>> . . . . page 1561

CGAL::operator>>

Function

Definition

This operator imports a polygon with holes, a general polygon, or a general polygon with holes from the input stream *in*.

An ASCII and a binary format exist. The stream detects the format automatically and can read both.

#include <CGAL/Polygon_with_holes_2.h> *template <class Kernel, Class Container>*

istream& *istream& in >> CGAL::Polygon_with_holes_2<Kernel, Container>& P*

The format consists of the number of points of the outer boundary followed by the points themselves in counterclockwise order, followed by the number of holes, and for each hole, the number of points of the outer boundary is followed by the points themselves in clockwise order.

#include <CGAL/General_polygon_2.h> *template <class ArrTraits>*

istream& *istream& in >> CGAL::General_polygon_2<ArrTraits>& P*

The format consists of the number of curves of the outer boundary followed by the curves themselves in counterclockwise order.

#include <CGAL/General_polygon_with_holes_2.h> *template <class Polygon>*

istream& *istream& in >> CGAL::General_polygon_with_holes_2<Polygon>& P*

The format consists of the number of curves of the outer boundary followed by the curves themselves in counterclockwise order, followed by the number of holes, and for each hole, the number of curves on its outer boundary is followed by the curves themselves in clockwise order.

See Also

CGAL::Polygon_2<PolygonTraits_2, Container> . . . . page 1472
CGAL::General_polygon_2<ArrTraits> . . . . page 1533
CGAL::General_polygon_with_holes_2<Polygon> . . . . page 1536
operator<< . . . . page 1560

Concept

GpsTraitsGeneralPolygon_2

Definition

A model of this concept represents a simple general polygon.

Refines

DefaultConstructible, CopyConstructible, Assignable.

Generalizes

GeneralPolygon_2

Has Models

GeneralPolygonSetTraits_2::Polygon_2 CGAL::Polygon_2<PolygonTraits_2, Container>

See Also

GeneralPolygon_2

GpsTraitsGeneralPolygonWithHoles_2

Definition

A model of this concept represents a general polygon with holes.

Refines

DefaultConstructible, CopyConstructible, Assignable.

Generalizes

GeneralPolygonWithHoles_2

Has Models

GeneralPolygonSetTraits_2::Polygon_with_holes2

See Also

GeneralPolygonWithHoles_2

Chapter 20

2D Boolean Operations on Nef Polygons

Michael Seel

Contents

20.1 Introduction

When working with polygonal and polyhedral sets, the mathematical model determines the kind of point set that can be represented. Nef polyhedra are the most general rectilinear polyhedral model.

Topological simpler models that are contained in the domain of Nef polyhedra are:

- *convex polytopes* normally defined as the convex hull of a nonempty finite set of points. Convex polytopes are compact closed and manifold sets.
- *elementary polyhedra* normally defined as the union of a finite number of convex polytopes.
- *polyhedral sets* nomally defined as the intersection of a finite number of closed halfspaces. Such sets are closed and convex but need not to be compact.
- *linear polyhedra* normally defined as the set of all points belonging to the simplices of a *simplicial complex*.

A planar *Nef polyhedron* is any set that can be obtained from a finite set of open halfspaces by set complement and set intersection operations. Due to the fact that all other binary set operations like union, difference and symmetric difference can be reduced to intersection and complement calculations, Nef polyhedra are also closed under those operations. Apart from the set complement operation there are more topological unary set operations that are closed in the domain of Nef polyhedra. Given a Nef polyhedron one can determine its interior, its boundary, and its closure, and also composed operations like regularization (defined to be the closure of the interior or a point set).

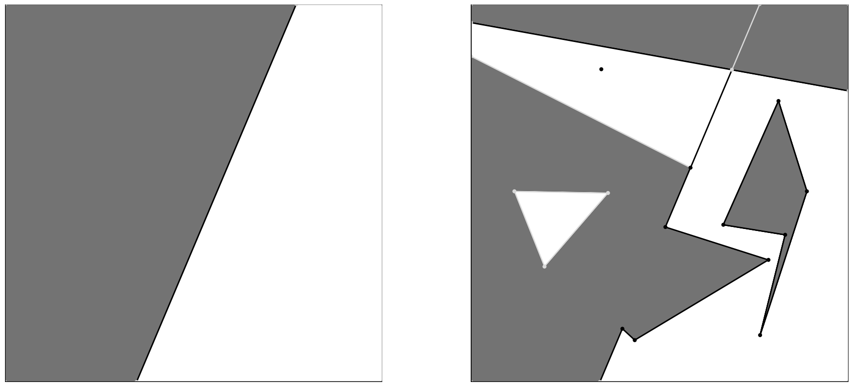


Figure 20.1: Two Nef polyhedra in the plane. A closed halfspace on the left and a complex polyhedron on the right. Note that the points on the squared boundary are at infinity.

20.2 Construction and Composition

Following the above definition, the data type *Nef_polyhedron_2<T>* allows construction of elementary Nef polyhedra and the binary and unary composition by the mentioned set operations.

In the following examples skip the typedefs at the beginning at first and take the types *Point* and *Line* to be models of the standard two-dimensional CGAL kernel (*CGAL::Point_2<K>* and *CGAL::Line_2<K>*). Their user interface is thus defined in the corresponding reference pages.

```
#include <CGAL/Gmpz.h>
#include <CGAL/Filtered_extended_homogeneous.h>
#include <CGAL/Nef_polyhedron_2.h>

typedef CGAL::Gmpz RT;
typedef CGAL::Filtered_extended_homogeneous<RT> Extended_kernel;
typedef CGAL::Nef_polyhedron_2<Extended_kernel> Nef_polyhedron;
typedef Nef_polyhedron::Point Point;
typedef Nef_polyhedron::Line  Line;

int main() {

  Nef_polyhedron N1(Nef_polyhedron::COMPLETE);

  Line l(2,4,2); // l : 2x + 4y + 2 = 0
  Nef_polyhedron N2(l,Nef_polyhedron::INCLUDED);
  Nef_polyhedron N3 = N2.complement();
  CGAL_assertion(N1 == N2.join(N3));

  Point p1(0,0), p2(10,10), p3(-20,15);
  Point triangle[3] = { p1, p2, p3 };
  Nef_polyhedron N4(triangle, triangle+3);
  Nef_polyhedron N5 = N2.intersection(N4);
```

```
  CGAL_assertion(N5 <= N2 && N5 <= N4);

  return 0;
}
```

File: examples/Nef_2/nef_2_construction.cpp

Planar halfspaces (as used in the definition) are modelled by oriented lines. In the previous example *N1* is the Nef polyhedron representing the full plane, *N2* is the closed halfspace left of the oriented line with equation $2x+4y+2=0$ including the line, *N3* is the complement of *N2* and therefore it must hold that $N2 \cup N3 = N1$.

Additionally one can construct Nef polyhedra from iterator ranges that hold simple polygonal chains. In the example *N4* is the triangle spanned by the vertices $(0,0)$, $(10,10)$, $(-20,15)$. Note that the construction from a simple polygonal chain has several cases and preconditions that are described in the reference manual page of *Nef_polyhedron_2<T>*. The *operator<=* in the last assertion is a subset-or-equal comparison of two polyhedra.

Nef polyhedra have input and output operators that allows one to output them via streams and read them from streams. Graphical output is currently possible. For an elaborate example see the demo programs in the directory *demo/Nef_2*.

20.3 Exploration

By recursively composing binary and unary operations one can end with a very complex rectilinear structure. To explore that structure there is a data type *Nef_polyhedron_2<T>::Explorer* that allows read-only exploration of the rectilinear structure. To understand its usability we need more knowledge about the representation of Nef polyhedra.

The rectilinear structure underlying a Nef polyhedron is stored in a selective plane map. Plane map here means a straightline embedded bidirected graph with face objects such that each point in the plane can be uniquely assigned to an object (vertex, edge, face) of the planar subdivision defined by the graph. Selective means that each object (vertex, edge, face) has a Boolean value associated with it to indicate set inclusion or exclusion.

The plane map is defined by the interface data type *Nef_polyhedron_2<T>::Topological_explorer*. Embedding the vertices by standard affine points does not suffice to model the unboundedness of halfspaces and ray-like structures. Therefore the planar subdivision is bounded symbolically by an axis-parallel square box of infimaximal size centered at the origin of our coordinate system. All structures extending to infinity are pruned by the box. Lines and rays have symbolic endpoints on the box. Faces are circularly closed. Infimaximal here means that its geometric extend is always large enough (but finite for our intuition). Assume you approach the box with an affine point, then this point is always inside the box. The same holds for straight lines; they always intersect the box. There are more accurate notions of “large enough”, but the previous propositions are enough at this point. Due to the fact that the infimaximal box is included in the plane map, the vertices and edges are partitioned with respect to this box.

Vertices inside the box are called standard vertices and they are embedded by affine points of type *Explorer::Point*. Vertices on the box are called non-standard vertices and they get their embedding where a ray intersects the box (their embedding is defined by an object of type *Explorer::Ray*). By their straightline embedding, edges represent either segments, rays, lines, or box segments depending on the character of their source and target vertices.

During exploration, box objects can be tracked down by the interface of *Nef_polyhedron_2<T>::Explorer* that is derived from *Nef_polyhedron_2<T>::Topological_explorer* and adds just the box exploration functionality to the interface of the latter. In the following code fragment we iterate over all vertices of a Nef polyhedron and check whether their embedding is an affine point or a point on the infimaximal frame.

```
typedef Nef_polyhedron::Explorer Explorer;
Explorer E = N4.explorer();
Explorer::Vertex_const_iterator v;
for (v = E.vertices_begin(); v != E.vertices_end(); ++v)
  if ( E.is_standard(v) )
    Explorer::Point p = E.point(v) // affine embedding of v
  else /* non-standard */
    Explorer::Ray r = E.ray(v) // extended embedding of v
```

Note that box edges only serve as boundary edges (combinatorically) to close the faces that extend to infinity (geometrically). Their status can be queried by the following operation:

```
typedef Nef_polyhedron::Explorer Explorer;
Explorer E = N4.explorer();
Explorer::Halfedge_const_iterator e;
for (e = E.halfedges_begin(); e != E.halfedges_end(); ++e)
  if ( E.is_frame_edge(e) ) // e is part of square box.
```

20.4 Traits Classes

Now finally we clarify what the template parameter of class *Nef_polyhedron_2<T>* actually models. *T* carries the implementation of a so-called extended geometric kernel.

Currently there are three kernel models: *CGAL::Extended_cartesian<FT>*, *CGAL::Extended_homogeneous<RT>*, and *CGAL::Filtered_extended_homogeneous<RT>*. The latter is the most optimized one. The former two are simpler versions corresponding to the simple planar affine kernels. Actually, it holds that (type equality in pseudo-code notation):

```
CGAL::Nef_polyhedron_2< CGAL::Extended_cartesian<FT> >::Point
  == CGAL::Cartesian<FT>::Point_2

CGAL::Nef_polyhedron_2< CGAL::Extended_homogeneous<RT> >::Point
   == CGAL::Homogeneous<RT>::Point_2

CGAL::Nef_polyhedron_2< CGAL::Filtered_extended_homogeneous<RT> >::Point
   == CGAL::Homogeneous<RT>::Point_2
```

Similar equations hold for the types *Line* and *Direction* in the local scope of *Nef_polyhedron_2<...>*.

advanced

For its notions and requirements see the desciption of the concept *ExtendedKernelTraits_2* in the reference manual.

advanced

20.5 Implementation

The underlying set operations are realized by an efficient and complete algorithm for the overlay of two plane maps. The algorithm is efficient in the sense that its running time is bounded by the size of the inputs plus the size of the output times a logarithmic factor. The algorithm is complete in the sense that it can handle all inputs and requires no general position assumption.

2D Boolean Operations on Nef Polygons Reference Manual

Michael Seel

Planar Nef Polyhedra are whatever can be constructed with a finite number of Boolean operations on halfspaces. See the user manual pages for a detailed introduction.

20.6 Classified Reference Pages

Concepts

Classes

20.7 Alphabetical List of Reference Pages

Class

CGAL::Nef_polyhedron_2<T>

Definition

An instance of data type *Nef_polyhedron_2<T>* is a subset of the plane that is the result of forming complements and intersections starting from a finite set *H* of halfspaces. *Nef_polyhedron_2* is closed under all binary set operations *intersection*, *union*, *difference*, *complement* and under the topological operations *boundary*, *closure*, and *interior*.

The template parameter *T* is specified via an extended kernel concept. *T* must be a model of the concept *ExtendedKernelTraits_2*.

#include <CGAL/Nef_polyhedron_2.h>

Types

Nef_polyhedron_2<T>:: Line	the oriented lines modeling halfplanes
Nef_polyhedron_2<T>:: Point	the affine points of the plane.
Nef_polyhedron_2<T>:: Direction	directions in our plane.
Nef_polyhedron_2<T>:: Polygons_tag	tag for calling polygon constructor.
Nef_polyhedron_2<T>:: Polylines_tag	tag for calling polyline constructor.
enum Boundary { EXCLUDED, INCLUDED};	construction selection.
enum Content { EMPTY, COMPLETE};	construction selection

Creation

Nef_polyhedron_2<T> N(Content plane = EMPTY);

creates an instance *N* of type *Nef_polyhedron_2<T>* and initializes it to the empty set if *plane == EMPTY* and to the whole plane if *plane == COMPLETE*.

Nef_polyhedron_2<T> N(Line l, Boundary line = INCLUDED);

creates a Nef polyhedron *N* containing the halfplane left of *l* including *l* if *line==INCLUDED*, excluding *l* if *line==EXCLUDED*.

template <class Forward_iterator>
Nef_polyhedron_2<T> N(Forward_iterator it, Forward_iterator end, Boundary b = INCLUDED);

creates a Nef polyhedron *N* from the simple polygon *P* spanned by the list of points in the iterator range *[it,end)* and including its boundary if *b = INCLUDED* excluding the boundary otherwise. *Forward_iterator* has to be an iterator with value type *Point*. This construction expects that *P* is simple. The degenerate cases where *P* contains no point, one point or spans just one segment (two points) are correctly handled. In all degenerate cases there's only one unbounded face adjacent to the degenerate polygon. If *b == INCLUDED* then *N* is just the boundary. If *b == EXCLUDED* then *N* is the whole plane without the boundary.

template <class Forward_iterator>
Nef_polyhedron_2<T> N(Forward_iterator it, Forward_iterator end, Polygons_tag);

The iterator range [it, end) defines a set point ranges, each of which defines the boundary of simple polygon.

template <class Forward_iterator>
Nef_polyhedron_2<T> N(Forward_iterator it, Forward_iterator end, Polylines_tag);

The iterator range [it, end) defines a set point ranges, each of which defines a polyline.

Operations

void	*N.clear(Content plane = EMPTY)*	makes *N* the empty set if *plane == EMPTY* and the full plane if *plane == COMPLETE*.
bool	*N.is_empty()*	returns true if *N* is empty, false otherwise.
bool	*N.is_plane()*	returns true if *N* is the whole plane, false otherwise.

Constructive Operations

Nef_polyhedron_2<T>	*N.complement()*	returns the complement of *N* in the plane.
Nef_polyhedron_2<T>	*N.interior()*	returns the interior of *N*.
Nef_polyhedron_2<T>	*N.closure()*	returns the closure of *N*.
Nef_polyhedron_2<T>	*N.boundary()*	returns the boundary of *N*.
Nef_polyhedron_2<T>	*N.regularization()*	returns the regularized polyhedron (closure of interior).

Nef_polyhedron_2<T>	*N.intersection(N1)*	returns $N \cap N1$.
Nef_polyhedron_2<T>	*N.join(N1)*	returns $N \cup N1$. Note that "union" is a keyword of C++ and cannot be used for this operation.
Nef_polyhedron_2<T>	*N.difference(N1)*	returns $N - N1$.
Nef_polyhedron_2<T>	*N.symmetric_difference(N1)*	returns the symmectric difference *N - T* $\cup$ *T - N*.

Additionally there are operators **,+,-,^,!* which implement the binary operations *intersection*, *join*, *difference*, *symmetric difference*, and the unary operation *complement*, respectively. There are also the corresponding modification operations **=,+=,-=,^=*.

There are also comparison operations like *<,<=,>,>=,==,!=* which implement the relations subset, subset or equal, superset, superset or equal, equality, inequality, respectively.

Exploration - Point location - Ray shooting

As Nef polyhedra are the result of forming complements and intersections starting from a set H of halfspaces that are defined by oriented lines in the plane, they can be represented by an attributed plane map $M = (V,E,F)$. For topological queries within M the following types and operations allow exploration access to this structure.

Types

Nef_polyhedron_2<T>:: Explorer

a decorator to examine the underlying plane map. See the manual page of *Explorer*.

Nef_polyhedron_2<T>:: Object_handle

a generic handle to an object of the underlying plane map. The kind of object *(vertex, halfedge, face)* can be determined and the object can be assigned to a corresponding handle by the three functions:
bool assign(Vertex_const_handle& h, Object_handle)
bool assign(Halfedge_const_handle& h, Object_handle)
bool assign(Face_const_handle& h, Object_handle)
where each function returns *true* iff the assignment to *h* was done.

enum Location_mode { *DEFAULT, NAIVE, LMWT*}*;*

selection flagfor the point location mode. LMWT stands for Locally Minimum Weight Triangulation, a locally optimized constrained triangulation where the weight corresponds to the length of the edges of the triangulation.

Operations

bool *N.contains(Object_handle h)*

returns true iff the object h is contained in the set represented by N.

bool *N.contained_in_boundary(Object_handle h)*

returns true iff the object h is contained in the 1-skeleton of N.

Object_handle *N.locate(Point p, Location_mode m = DEFAULT)*

returns a generic handle h to an object (face, halfedge, vertex) of the underlying plane map that contains the point p in its relative interior. The point p is contained in the set represented by N if *N.contains(h)* is true. The location mode flag m allows one to choose between different point location strategies.

Object_handle *N.ray_shoot(Point p, Direction d, Location_mode m = DEFAULT)*

returns a handle h with *N.contains(h)*, that can be converted to a *Vertex_/Halfedge_/Face_const_handle* as described above. The object returned is intersected by the ray starting in p with direction d and has minimal distance to p. The operation returns an empty *Object_handle* if the ray shoot along d does not hit any object h of N with *N.contains(h)*. The location mode flag m allows one to choose between different point location strategies.

Object_handle *N.ray_shoot_to_boundary(Point p, Direction d, Location_mode m = DEFAULT)*

returns a handle h, that can be converted to a *Vertex_/Halfedge_const_handle* as described above. The object returned is part of the 1-skeleton of N, intersected by the ray starting in p with direction d and has minimal distance to p. The operation returns an empty *Object_handle* if the ray shoot along d does not hit any 1-skeleton object h of N. The location mode flag m allows one to choose between different point location strategies.

Explorer *N.explorer()*

returns a decorator object that allows read-only access of the underlying plane map. See the manual page *Explorer* for its usage.

Implementation

Nef polyhedra are implemented on top of a halfedge data structure and use linear space in the number of vertices, edges and facets. Operations like *empty* take constant time. The operations *clear*, *complement*, *interior*, *closure*, *boundary*, *regularization*, input and output take linear time. All binary set operations and comparison operations take time $O(n \log n)$ where n is the size of the output plus the size of the input.

The point location and ray shooting operations are implemented in two flavors. The *NAIVE* operations run in linear query time without any preprocessing, the *DEFAULT* operations (equals *LMWT*) run in sub-linear query time, but preprocessing is triggered with the first operation. Preprocessing takes time $O(N^2)$, the sub-linear point location time is either logarithmic when LEDA's persistent dictionaries are present or if not then the point location time is worst-case linear, but experiments show often sublinear runtimes. Ray shooting equals point location plus a walk in the constrained triangulation overlayed on the plane map representation. The cost of the walk is proportional to the number of triangles passed in direction d until an obstacle is met. In a minimum weight triangulation of the obstacles (the plane map representing the polyhedron) the theory provides a $O(\sqrt{n})$ bound for the number of steps. Our locally minimum weight triangulation approximates the minimum weight triangulation only heuristically (the calculation of the minimum weight triangulation is conjectured to be NP hard). Thus we have no runtime guarantee but a strong experimental motivation for its approximation.

Example

Nef polyhedra are parameterized by a so-called extended geometric kernel. There are three kernels, one based on a homogeneous representation of extended points called *Extended_homogeneous<RT>* where *RT* is a ring type providing additionally a *gcd* operation, one based on a Cartesian representation of extended points called *Extended_cartesian<NT>* where *NT* is a field type, and finally *Filtered_extended_homogeneous<RT>* (an optimized version of the first). The following example uses the filtered homogeneous kernel to construct the intersection of two halfspaces.

```
#include <CGAL/Gmpz.h>
#include <CGAL/Filtered_extended_homogeneous.h>
#include <CGAL/Nef_polyhedron_2.h>

typedef CGAL::Gmpz RT;
typedef CGAL::Filtered_extended_homogeneous<RT> Extended_kernel;
typedef CGAL::Nef_polyhedron_2<Extended_kernel> Nef_polyhedron;
typedef Nef_polyhedron::Line  Line;

int main()
{
  Nef_polyhedron N1(Line(1,0,0));
  Nef_polyhedron N2(Line(0,1,0), Nef_polyhedron::EXCLUDED);
  Nef_polyhedron N3 = N1 * N2; // line (*)
  return 0;
}
```

File: `examples/Nef_2/nef_2_intersection.cpp`

After line (*) *N3* is the intersection of *N1* and *N2*. The member types of *Nef_polyhedron_2< Extended_homogeneous<NT> >* map to corresponding types of the standard CGAL geometry kernel (type equality in pseudo-code notation):

```
CGAL::Nef_polyhedron_2< CGAL::Extended_cartesian<FT> >::Point
  == CGAL::Cartesian<FT>::Point_2

CGAL::Nef_polyhedron_2< CGAL::Extended_homogeneous<RT> >::Point
   == CGAL::Homogeneous<RT>::Point_2

CGAL::Nef_polyhedron_2< CGAL::Filtered_extended_homogeneous<RT> >::Point
   == CGAL::Homogeneous<RT>::Point_2
```

The same holds for the types *Line* and *Direction* in the local scope of *Nef_polyhedron_2<...>*.

Class

CGAL::Topological_explorer

Definition

An instance D of the data type *Topological_explorer* is a decorator for interfacing the topological structure of a plane map P (read-only).

A plane map P consists of a triple (V,E,F) of vertices, edges, and faces. We collectively call them objects. An edge e is a pair of vertices *(v,w)* with incidence operations v = *source(e)*, w = *target(e)*. The list of all edges with source v is called the adjacency list *A(v)*.

Edges are paired into twins. For each edge e = *(v,w)* there's an edge *twin(e)* = *(w,v)* and *twin(twin(e))* == e[1].

An edge e = *(v,w)* knows two adjacent edges *en* = *next(e)* and *ep* = *previous(e)* where *source(en)* = w, *previous(en)* = e and *target(ep)* = v and *next(ep)* = e. By this symmetric *previous-next* relationship all edges are partitioned into face cycles. Two edges e and e' are in the same face cycle if $e = next^*(e')$. All edges e in the same face cycle have the same incident face $f = face(e)$. The cyclic order on the adjacency list of a vertex v = *source(e)* is given by *cyclic_adj_succ(e)* = *twin(previous(e))* and *cyclic_adj_pred(e)* = *next(twin(e))*.

A vertex v is embedded via coordinates *point(v)*. By the embedding of its source and target an edge corresponds to a segment. P has the property that the embedding is always *order-preserving*. This means a ray fixed in *point(v)* of a vertex v and swept around counterclockwise meets the embeddings of *target(e)* ($e \in A(v)$) in the cyclic order defined by the list order of A.

The embedded face cycles partition the plane into maximal connected subsets of points. Each such set corresponds to a face. A face is bounded by its incident face cycles. For all the edges in the non-trivial face cycles it holds that the face is left of the edges. There can also be trivial face cycles in form of isolated vertices in the interior of a face. Each such vertex v knows its surrounding face f = *face(v)*.

Plane maps are attributed, for each object $u \in V \cup E \cup F$ we attribute an information *mark(u)* of type *Mark*. *Mark* fits the concepts assignable, default-constructible, and equal-comparable.

Types

Topological_explorer:: Plane_map	The underlying plane map type
Topological_explorer:: Point	The point type of vertices.
Topological_explorer:: Mark	All objects (vertices, edges, faces) are attributed by a *Mark* object.
Topological_explorer:: Size_type	The size type.

Local types are handles, iterators and circulators of the following kind: *Vertex_const_handle*, *Vertex_const_iterator*, *Halfedge_const_handle*, *Halfedge_const_iterator*, *Face_const_handle*, *Face_const_iterator*. Additionally the following circulators are defined.

Topological_explorer:: Halfedge_around_vertex_const_circulator

circulating the outgoing halfedges in $A(v)$.

[1]The existence of the edge pairs makes P a bidirected graph, the *twin* links make P a map.

Topological_explorer:: Halfedge_around_face_const_circulator

circulating the halfedges in the face cycle of a face f.

Topological_explorer:: Hole_const_iterator iterating all holes of a face f. The type is convertible to *Halfedge_const_handle*.

Topological_explorer:: Isolated_vertex_const_iterator

iterating all isolated vertices of a face f. The type generalizes *Vertex_const_handle*.

Operations

Vertex_const_handle *D.source(Halfedge_const_handle e)*

returns the source of e.

Vertex_const_handle *D.target(Halfedge_const_handle e)*

returns the target of e.

Halfedge_const_handle *D.twin(Halfedge_const_handle e)*

returns the twin of e.

bool *D.is_isolated(Vertex_const_handle v)*

returns *true* iff $A(v) = \emptyset$.

Halfedge_const_handle *D.first_out_edge(Vertex_const_handle v)*

returns one halfedge with source v. It's the starting point for the circular iteration over the halfedges with source v.
Precondition: *!is_isolated(v)*.

Halfedge_const_handle *D.last_out_edge(Vertex_const_handle v)*

returns the halfedge with source v that is the last in the circular iteration before encountering *first_out_edge(v)* again.
Precondition: *!is_isolated(v)*.

Halfedge_const_handle *D.cyclic_adj_succ(Halfedge_const_handle e)*

returns the edge after e in the cyclic ordered adjacency list of *source(e)*.

Halfedge_const_handle	*D.cyclic_adj_pred(Halfedge_const_handle e)*	
		returns the edge before *e* in the cyclic ordered adjacency list of *source(e)*.
Halfedge_const_handle	*D.next(Halfedge_const_handle e)*	
		returns the next edge in the face cycle containing *e*.
Halfedge_const_handle	*D.previous(Halfedge_const_handle e)*	
		returns the previous edge in the face cycle containing *e*.
Face_const_handle	*D.face(Halfedge_const_handle e)*	
		returns the face incident to *e*.
Face_const_handle	*D.face(Vertex_const_handle v)*	
		returns the face incident to *v*. *Precondition*: *is_isolated(v)*.
Halfedge_const_handle	*D.halfedge(Face_const_handle f)*	
		returns a halfedge in the bounding face cycle of *f* (*Halfedge_const_handle()* if there is no bounding face cycle).

Iteration

Vertex_const_iterator	*D.vertices_begin()*	iterator over vertices of the map.
Vertex_const_iterator	*D.vertices_end()*	past-the-end iterator for vertices.
Halfedge_const_iterator	*D.halfedges_begin()*	iterator over halfedges of the map.
Halfedge_const_iterator	*D.halfedges_end()*	past-the-end iterator for halfedges.
Face_const_iterator	*D.faces_begin()*	iterator over faces of the map.
Face_const_iterator	*D.faces_end()*	past-the-end iterator for faces

Halfedge_around_vertex_const_circulator

D.out_edges(Vertex_const_handle v)

returns a circulator for the cyclic adjacency list of v.

Halfedge_around_face_const_circulator

D.face_cycle(Face_const_handle f)

returns a circulator for the outer face cycle of f.

Hole_const_iterator *D.holes_begin(Face_const_handle f)*

returns an iterator for all holes in the interior of f. A *Hole_iterator* can be assigned to a *Halfedge_around_face_const_circulator*.

Hole_const_iterator *D.holes_end(Face_const_handle f)*

returns the past-the-end iterator of f.

Isolated_vertex_const_iterator

D.isolated_vertices_begin(Face_const_handle f)

returns an iterator for all isolated vertices in the interior of f.

Isolated_vertex_const_iterator

D.isolated_vertices_end(Face_const_handle f)

returns the past the end iterator of f.

Associated Information

The type *Mark* is the general attribute of an object.

Point *D.point(Vertex_const_handle v)*

returns the embedding of v.

Mark *D.mark(Vertex_const_handle v)*

returns the mark of v.

Mark *D.mark(Halfedge_const_handle e)*

returns the mark of e.

Mark *D.mark(Face_const_handle f)*

returns the mark of f.

Statistics and Integrity

Size_type *D.number_of_vertices()* returns the number of vertices.

Size_type *D.number_of_halfedges()* returns the number of halfedges.

Size_type *D.number_of_edges()* returns the number of halfedge pairs.

Size_type *D.number_of_faces()* returns the number of faces.

Size_type *D.number_of_face_cycles()* returns the number of face cycles.

Size_type *D.number_of_connected_components()*

calculates the number of connected components of P.

void *D.print_statistics(std::ostream& os = std::cout)*

print the statistics of P: the number of vertices, edges, and faces.

void *D.check_integrity_and_topological_planarity(bool faces=true)*

checks the link structure and the genus of P.

Class

CGAL::Explorer

Definition

An instance E of the data type *Explorer* is a decorator to explore the structure of the plane map underlying the Nef polyhedron. It inherits all topological adjacency exploration operations from *Topological_explorer*. *Explorer* additionally allows one to explore the geometric embedding.

The position of each vertex is given by a so-called extended point, which is either a standard affine point or the tip of a ray touching an infinimaximal square frame centered at the origin. A vertex v is called a *standard* vertex if its embedding is a *standard* point and *non-standard* if its embedding is a *non-standard* point. By the straightline embedding of their source and target vertices, edges correspond to either affine segments, rays or lines or are part of the bounding frame.

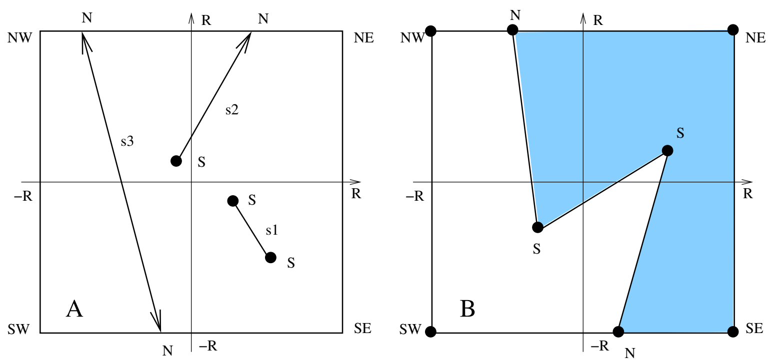


Figure 20.2: Extended geometry: standard vertices are marked by S, non-standard vertices are marked by N. **A**: The possible embeddings of edges: an affine segment s1, an affine ray s2, an affine line s3. **B**: A plane map embedded by extended geometry: note that the frame is arbitrarily large, the 6 vertices on the frame are at infinity, the two faces represent a geometrically unbounded area, however they are topologically closed by the frame edges. No standard point can be placed outside the frame.

Inherits From

Topological_explorer

Types

Explorer:: Point	the point type of finite vertices.
Explorer:: Ray	the ray type of vertices on the frame.

Iterators, handles, and circulators are inherited from *Topological_explorer*.

Creation

Explorer is copy constructable and assignable. An object can be obtained via the *Nef_polyhedron_2::explorer()* method of *Nef_polyhedron_2*.

Operations

bool *E.is_standard(Vertex_const_handle v)*

returns true iff *v*'s position is a standard point.

Point *E.point(Vertex_const_handle v)*

returns the standard point that is the embedding of *v*.
Precondition: *E.is_standard(v)*.

Ray *E.ray(Vertex_const_handle v)*

returns the ray defining the non-standard point on the frame.
Precondition: *!E.is_standard(v)*.

bool *E.is_frame_edge(Halfedge_const_handle e)*

returns true iff *e* is part of the infinimaximal frame.

ExtendedKernelTraits_2

Concept

Definition

ExtendedKernelTraits_2 is a kernel concept providing extended geometry[2]. Let K be an instance of the data type *ExtendedKernelTraits_2*. The central notion of extended geometry are extended points. An extended point represents either a standard affine point of the Cartesian plane or a non-standard point representing the equivalence class of rays where two rays are equivalent if one is contained in the other.

Let R be an infinimaximal number[3], F be the square box with corners $NW(-R,R)$, $NE(R,R)$, $SE(R,-R)$, and $SW(-R,-R)$. Let p be a non-standard point and let r be a ray defining it. If the frame F contains the source point of r then let $p(R)$ be the intersection of r with the frame F, if F does not contain the source of r then $p(R)$ is undefined. For a standard point let $p(R)$ be equal to p if p is contained in the frame F and let $p(R)$ be undefined otherwise. Clearly, for any standard or non-standard point p, $p(R)$ is defined for any sufficiently large R. Let f be any function on standard points, say with k arguments. We call f *extensible* if for any k points $p_1, \ldots, p_k$ the function value $f(p_1(R), \ldots, p_k(R))$ is constant for all sufficiently large R. We define this value as $f(p_1, \ldots, p_k)$. Predicates like lexicographic order of points, orientation, and incircle tests are extensible.

An extended segment is defined by two extended points such that it is either an affine segment, an affine ray, an affine line, or a segment that is part of the square box. Extended directions extend the affine notion of direction to extended objects.

This extended geometry concept serves two purposes. It offers functionality for changing between standard affine and extended geometry. At the same time it provides extensible geometric primitives on the extended geometric objects.

Types

Affine kernel types

ExtendedKernelTraits_2:: Standard_kernel	the standard affine kernel.
ExtendedKernelTraits_2:: Standard_RT	the standard ring type.
ExtendedKernelTraits_2:: Standard_point_2	standard points.
ExtendedKernelTraits_2:: Standard_segment_2	standard segments.
ExtendedKernelTraits_2:: Standard_line_2	standard oriented lines.
ExtendedKernelTraits_2:: Standard_direction_2	standard directions.
ExtendedKernelTraits_2:: Standard_ray_2	standard rays.

[2] It is called extended geometry for simplicity, though it is not a real geometry in the classical sense.
[3] A finite but very large number.

ExtendedKernelTraits_2:: Standard_aff_transformation_2

standard affine transformations.

Extended kernel types

ExtendedKernelTraits_2:: RT — the ring type of our extended kernel.

ExtendedKernelTraits_2:: Point_2 — extended points.

ExtendedKernelTraits_2:: Segment_2 — extended segments.

ExtendedKernelTraits_2:: Direction_2 — extended directions.

*enum Point_type { SWCORNER,
LEFTFRAME,
NWCORNER,
BOTTOMFRAME,
STANDARD,
TOPFRAME,
SECORNER,
RIGHTFRAME,
NECORNER}*

a type descriptor for extended points.

Operations

Interfacing the affine kernel types

Point_2 *K.construct_point(Standard_point_2 p)*

creates an extended point and initializes it to the standard point *p*.

Point_2 *K.construct_point(Standard_line_2 l)*

creates an extended point and initializes it to the equivalence class of all the rays underlying the oriented line *l*.

Point_2 *K.construct_point(Standard_point_2 p1, Standard_point_2 p2)*

creates an extended point and initializes it to the equivalence class of all the rays underlying the oriented line *l(p1,p2)*.

Point_2	*K.construct_point(Standard_point_2 p, Standard_direction_2 d)*	
		creates an extended point and initializes it to the equivalence class of all the rays underlying the ray starting in *p* in direction *d*.
Point_2	*K.construct_opposite_point(Standard_line_2 l)*	
		creates an extended point and initializes it to the equivalence class of all the rays underlying the oriented line opposite to *l*.
Point_type	*K.type(Point_2 p)*	determines the type of *p* and returns it.
bool	*K.is_standard(Point_2 p)*	
		returns *true* iff *p* is a standard point.
Standard_point_2	*K.standard_point(Point_2 p)*	
		returns the standard point represented by *p*. *Precondition*: *K.is_standard(p)*.
Standard_line_2	*K.standard_line(Point_2 p)*	
		returns the oriented line representing the bundle of rays defining *p*. *Precondition*: *!K.is_standard(p)*.
Standard_ray_2	*K.standard_ray(Point_2 p)*	
		a ray defining *p*. *Precondition*: *!K.is_standard(p)*.
Point_2	*K.NE()*	returns the point on the northeast frame corner.
Point_2	*K.SE()*	returns the point on the southeast frame corner.
Point_2	*K.NW()*	returns the point on the northwest frame corner.
Point_2	*K.SW()*	returns the point on the southwest frame corner.

Geometric kernel calls

Point_2	*K.source(Segment_2 s)*	
		returns the source point of *s*.

Point_2 *K.target(Segment_2 s)*

returns the target point of *s*.

Segment_2 *K.construct_segment(Point_2 p, Point_2 q)*

constructs a segment *pq*.

int *K.orientation(Segment_2 s, Point_2 p)*

returns the orientation of *p* with respect to the line through *s*.

int *K.orientation(Point_2 p1, Point_2 p2, Point_2 p3)*

returns the orientation of *p3* with respect to the line through *p1p2*.

bool *K.left_turn(Point_2 p1, Point_2 p2, Point_2 p3)*

return true iff the *p3* is left of the line through *p1p2*.

bool *K.is_degenerate(Segment_2 s)*

return true iff *s* is degenerate.

int *K.compare_xy(Point_2 p1, Point_2 p2)*

returns the lexicographic order of *p1* and *p2*.

int *K.compare_x(Point_2 p1, Point_2 p2)*

returns the order on the *x*-coordinates of *p1* and *p2*.

int *K.compare_y(Point_2 p1, Point_2 p2)*

returns the order on the *y*-coordinates of *p1* and *p2*.

Point_2 *K.intersection(Segment_2 s1, Segment_2 s2)*

returns the point of intersection of the lines supported by *s1* and *s2*.
Precondition: the intersection point exists.

Direction_2	*K.construct_direction(Point_2 p1, Point_2 p2)*	
		returns the direction of the vector *p2 - p1*.
bool	*K.strictly_ordered_ccw(Direction_2 d1, Direction_2 d2, Direction_2 d3)*	
		returns *true* iff *d2* is in the interior of the counterclockwise angular sector between *d1* and *d3*.
bool	*K.strictly_ordered_along_line(Point_2 p1, Point_2 p2, Point_2 p3)*	
		returns *true* iff *p2* is in the relative interior of the segment *p1p3*.
bool	*K.contains(Segment_2 s, Point_2 p)*	
		returns true iff *s* contains *p*.
bool	*K.first_pair_closer_than_second(Point_2 p1, Point_2 p2, Point_2 p3, Point_2 p4)*	
		returns true iff $\|p1-p2\| < \|p3-p4\|$.
*const char**	*K.output_identifier()*	returns a unique identifier for kernel object Input/Output. Usually this should be the name of the model.

Has Models

Class

CGAL::Extended_cartesian<FT>

Definition

The class *Extended_cartesian<FT>* serves as a traits class for the class *CGAL::Nef_polyhedron_2<T>*. It uses a polynomial component representation based on a field number type *FT*.

#include <CGAL/Extended_cartesian.h>

Is Model for the Concepts

ExtendedKernelTraits_2 . . . . page 1585

Creation

Extended_cartesian<FT> traits;	default constructor.

Requirements

To make a field number type *FT_model* work with this class, you must provide a traits class for this number type: *CGAL::Number_type_traits<FT_model>* (See the support library manual.)

Operations

Fits all operation requirements of the concept.

See Also

CGAL::Extended_homogeneous<RT> . . . . page 1591
CGAL::Filtered_extended_homogeneous<RT> . . . . page 1592

CGAL::Extended_homogeneous<RT>

Class

Definition

The class *Extended_homogeneous<RT>* serves as a traits class for the class *CGAL::Nef_polyhedron_2<T>*. It uses a polynomial component representation based on a Euclidean ring number type *RT*.

#include <CGAL/Extended_homogeneous.h>

Is Model for the Concepts

ExtendedKernelTraits_2 . . . . page 1585

Creation

Extended_homogeneous<RT> traits;	default constructor.

Requirements

To make an Euclidean ring number type *RT_model* work with this class the number type must support a gcd computation in namespace *CGAL::NTS*. CGAL provides a function template for this, which will be used by default when your number type is not one of the built-in number types, one of the number types distrubuted with CGAL or one of the LEDA number types.

Operations

Fits all operation requirements of the concept.

See Also

CGAL::Extended_cartesian<FT> . . . . page 1590
CGAL::Filtered_extended_homogeneous<RT> . . . . page 1592

Class

CGAL::Filtered_extended_homogeneous<RT>

Definition

The class *Filtered_extended_homogeneous<RT>* serves as a traits class for the class *CGAL::Nef_polyhedron_2<T>*. It uses a polynomial component representation based on a ring number type *RT*.

#include <CGAL/Filtered_extended_homogeneous.h>

Is Model for the Concepts

ExtendedKernelTraits_2 . . . . page 1585

Creation

Filtered_extended_homogeneous<RT> traits; default constructor.

Operations

Fits all operation requirements of the concept.

See Also

CGAL::Extended_cartesian<FT> . . . . page 1590
CGAL::Extended_homogeneous<RT> . . . . page 1591

Chapter 21

2D Boolean Operations on Nef Polygons Embedded on the Sphere

Peter Hachenberger and Lutz Kettner

Contents

21.1 Introduction

Nef polyhedra are defined as a subset of the d-dimensional space obtained by a finite number of set complement and set intersection operations on halfspaces.

Due to the fact that all other binary set operations like union, difference and symmetric difference can be reduced to intersection and complement calculations, Nef polyhedra are also closed under those operations. Also, Nef polyhedra are closed under topological unary set operations. Given a Nef polyhedron one can determine its interior, its boundary, and its closure.

Additionally, a d-dimensional Nef polyhedron has the property, that its boundary is a (d-1)-dimensional Nef polyhedron. This property can be used as a way to represent 3-dimensional Nef polyhedra by means of planar Nef polyhedra. This is done by intersecting the neighborhood of a vertex in a 3D Nef polyhedron with an ε-sphere. The result is a planar Nef polyhedron embedded on the sphere.

The intersection of a halfspace going through the center of the ε-sphere, with the ε-sphere, results in a halfsphere which is bounded by a great circle. A binary operation of two halfspheres cuts the great circles into great arcs.

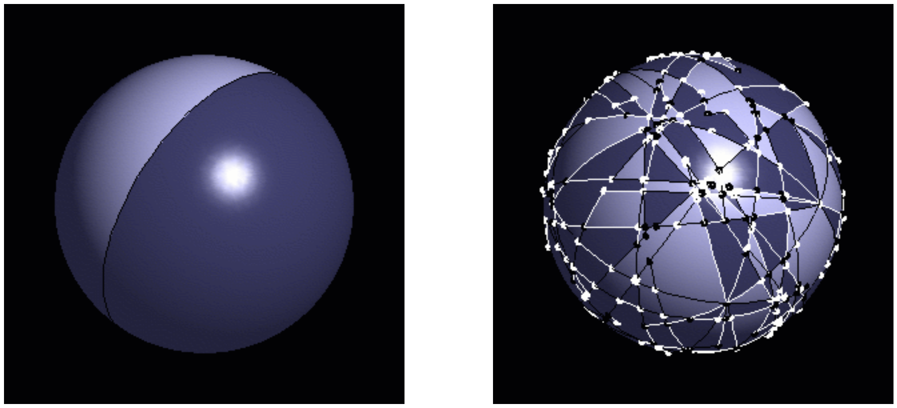
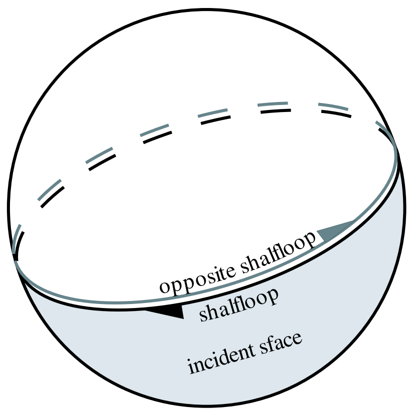


Figure 21.1: Two spherical Nef polyhedra. A closed halfspace on the left and a complex polyhedron on the right. The different colors indicate selected and unselected regions, lines and points.



The incidence structure of planar Nef polyhedra can be reused. The items are denoted as $svertex$, $shalfedge$ and $sface$, analogous to their counterparts in *Nef_polyhedron_S2*. Additionally, there is the *shalfloop* reprsenting the great circles. The incidences are illustrated in the figure above.

21.2 Restricted Spherical Geometry

We introduce geometric objects that are part of the spherical surface S_2 and operations on them. We define types *Sphere_point*, *Sphere_circle*, *Sphere_segment*, and *Sphere_direction*. *Sphere_point*s are points on S_2, *Sphere_circle*s are oriented great circles of S_2, *Sphere_segment*s are oriented parts of *Sphere_circles* bounded by a pair of *Sphere_point*s, and *Sphere_direction*s are directions that are part of great circles. (a direction is usually defined to be a vector without length, that floats around in its underlying space and can be used to specify a movement at any point of the underlying space; in our case we use directions only at points that are part of the great circle that underlies also the direction.)

Note that we have to consider special geometric properties of the objects. For example two points that are part of a great circle define two *Sphere_segment*s, and two arbitrary *Sphere_segment*s can intersect in two points.

If we restrict our geometric objects to a so-called perfect hemisphere of S_2[1] then the restricted objects behave like in classical geometry, e.g., two points define exactly one segment, two segments intersect in at most one

[1] A perfect hemisphere of S_2 is an open half-sphere plus an open half-circle in the boundary of the open half-sphere plus one endpoint of the half-circle.

interior point (non-degenerately), or three non-cocircular sphere points can be qualified as being positively or negatively oriented.

21.3 Example Programs

21.3.1 First Example

In this first example *Nef_polyhedron_S2* is parametrized with a CGAL Kernel as traits class. The types comprising the spherical geometry can be retrieved from the type *Nef_polyhedron_S2<Traits>* as is done in the example with the type *Sphere_circle*. Then three Nef polyhedra are created: $N1$ is a halfsphere including the boundary, $N2$ is another halfsphere without the boundary, and $N3$ is the intersection of $N1$ and $N2$.

```
#include <CGAL/Gmpz.h>
#include <CGAL/Homogeneous.h>
#include <CGAL/Nef_polyhedron_S2.h>

typedef CGAL::Gmpz RT;
typedef CGAL::Homogeneous<RT> Kernel;
typedef CGAL::Nef_polyhedron_S2<Kernel> Nef_polyhedron;
typedef Nef_polyhedron::Sphere_circle Sphere_circle;

int main()
{
  Nef_polyhedron N1(Sphere_circle(1,0,0));
  Nef_polyhedron N2(Sphere_circle(0,1,0), Nef_polyhedron::EXCLUDED);
  Nef_polyhedron N3 = N1 * N2;
  return 0;
}
```

File: examples/Nef_S2/nef_s2_simple.cpp

21.3.2 Construction and Combinations

Th example shows the different types of constructors: $N1$ is the complete sphere, $N2$ is a halfsphere which includes the boundary, $N3$ is created with the copy constructor, $N4$ is created as an arrangement of a set of *Sphere_segments*, and $N5$ is created as the empty set.

The example also shows the use of unary set operations, binary operations, and binary predicates: $N3$ is defined as the complement of $N2$, $N1$ is compared with the union of $N2$ and $N3$, $N5$ is united with $N2$ and then intersected with $N4$. At last, it is tested if $N5$ is a subset of $N2$ and if $N5$ is not equal to $N4$.

```
#include <CGAL/Gmpz.h>
#include <CGAL/Homogeneous.h>
#include <CGAL/Nef_polyhedron_S2.h>

typedef CGAL::Gmpz RT;
typedef CGAL::Homogeneous<RT> Kernel;
typedef CGAL::Nef_polyhedron_S2<Kernel> Nef_polyhedron;
typedef Nef_polyhedron::Sphere_point Sphere_point;
```

```
typedef Nef_polyhedron::Sphere_segment Sphere_segment;
typedef Nef_polyhedron::Sphere_circle Sphere_circle;

int main() {

  Nef_polyhedron N1(Nef_polyhedron::COMPLETE);

  Sphere_circle c(1,1,1); // c : x + y + z = 0
  Nef_polyhedron N2(c, Nef_polyhedron::INCLUDED);
  Nef_polyhedron N3(N2.complement());
  CGAL_assertion(N1 == N2.join(N3));

  Sphere_point   p1(1,0,0), p2(0,1,0), p3(0,0,1);
  Sphere_segment s1(p1,p2), s2(p2,p3), s3(p3,p1);
  Sphere_segment triangle[3] = { s1, s2, s3 };
  Nef_polyhedron N4(triangle, triangle+3);
  Nef_polyhedron N5;
  N5 += N2;
  N5 = N5.intersection(N4);
  CGAL_assertion(N5 <= N2 && N5 != N4);

  return 0;
}
```

File: examples/Nef_S2/nef_s2_construction.cpp

21.3.3 Exploration

By recursively composing binary and unary operations one can end with a very complex rectilinear structure. *Nef_polyhedron_S2* allows read-only exploration of the structure.

In the following example, a random *Nef_polyhedron_S2 S* created from *n* halfspheres is explored. Each sface is composed of one outer sface cycles and an arbitrary number of inner sfaces cycles. The outer cycle is either an shalfloop or a cycle of shalfedges. An inner cycles additionally can be an isolated vertex. The example shows how to get the entry item *it* to all sface cycles of an sface *sf* and how to find out what type of item it is.

The macro *CGAL_forall_sface_cycles_of* is equivalent to a for-loop on the range *[sf->sface_cycles_begin(), sf->sface_cycles_end())*. An *SFace_cycle_const_iterator* either represents a *SVertex_const_handle*, a *SHalfede_const_handle* or a *SHalfloop_const_handle*. In order to find out which handle type is represented, the functions *is_svertex()*, *is_shafledge()* and *is_shalfloop()* are provided. Afterwards the iterator can be casted to the proper handle type.

```
#include <CGAL/Gmpq.h>
#include <CGAL/Cartesian.h>
#include <CGAL/Nef_polyhedron_S2.h>
#include <CGAL/Nef_S2/create_random_Nef_S2.h>

typedef CGAL::Gmpq FT;
typedef CGAL::Cartesian<FT> Kernel;
typedef CGAL::Nef_polyhedron_S2<Kernel> Nef_polyhedron_S2;
typedef Nef_polyhedron_S2::SVertex_const_handle SVertex_const_handle;
typedef Nef_polyhedron_S2::SHalfedge_const_handle SHalfedge_const_handle;
```

```
typedef Nef_polyhedron_S2::SHalfloop_const_handle SHalfloop_const_handle;
typedef Nef_polyhedron_S2::SFace_const_iterator SFace_const_iterator;
typedef Nef_polyhedron_S2::SFace_cycle_const_iterator
                            SFace_cycle_const_iterator;

int main() {

  Nef_polyhedron_S2 S;
  CGAL::create_random_Nef_S2(S,5);

  int i=0;
  SFace_const_iterator sf;
  CGAL_forall_sfaces(sf,S) {
    SFace_cycle_const_iterator it;
    std::cout << "the sface cycles of sface " << i++;
    std::cout << " start with an " << std::endl;
    CGAL_forall_sface_cycles_of(it,sf) {
      if (it.is_svertex()) {
        std::cout << "  svertex at position ";
        std::cout << SVertex_const_handle(it)->point() << std::endl;
      }
      else if (it.is_shalfedge()) {
        std::cout << "  shalfedge from ";
        std::cout << SHalfedge_const_handle(it)->source()->point() << "
   to ";
        std::cout << SHalfedge_const_handle(it)->target()->point() <<
   std::endl;
      }
      else if (it.is_shalfloop()) {
        std::cout << "  shalfloop lying in the plane ";
        std::cout << SHalfloop_const_handle(it)->circle() << std::endl;
      }
      else
        std::cout << "something is wrong" << std::endl;
    }
  }
  return 0;
}
```

File: examples/Nef_S2/nef_s2_exploration.cpp

21.3.4 Point Location

Using the *locate* function, it is possible to retrive an item at a certain location on the sphere. In the following example, the item at location *Sphere_point(1,0,0)* in a random *Nef_polyhedron_S2* is retrieved. *locate* returns an instance of type *Object_handle*, which is a container for any handle type. Here, it either a *SVertex_const_handle*, a *SHalfedge_const_handle*, a *SHafloop_const_handle* or a *SFace_const_handle*. The function *CGAL::assign* performs the cast operation and returns a boolean which indicates whether the cast was successful or not.

```
#include <CGAL/Gmpq.h>
#include <CGAL/Cartesian.h>
#include <CGAL/Nef_polyhedron_S2.h>
```

```
#include <CGAL/Nef_S2/create_random_Nef_S2.h>

typedef CGAL::Gmpq FT;
typedef CGAL::Cartesian<FT> Kernel;
typedef CGAL::Nef_polyhedron_S2<Kernel> Nef_polyhedron_S2;
typedef Nef_polyhedron_S2::SVertex_const_handle SVertex_const_handle;
typedef Nef_polyhedron_S2::SHalfedge_const_handle SHalfedge_const_handle;
typedef Nef_polyhedron_S2::SHalfloop_const_handle SHalfloop_const_handle;
typedef Nef_polyhedron_S2::SFace_const_handle SFace_const_handle;
typedef Nef_polyhedron_S2::Object_handle Object_handle;
typedef Nef_polyhedron_S2::Sphere_point Sphere_point;

int main() {

  Nef_polyhedron_S2 S;
  CGAL::create_random_Nef_S2(S,5);

  SVertex_const_handle sv;
  SHalfedge_const_handle se;
  SHalfloop_const_handle sl;
  SFace_const_handle sf;
  Object_handle o = S.locate(Sphere_point(1,0,0));
  if(CGAL::assign(sv,o))
    std::cout << "Locating svertex" << std::endl;
  else if(CGAL::assign(se,o))
    std::cout << "Locating shalfedge" << std::endl;
  else if(CGAL::assign(sl,o))
    std::cout << "Locating shalfloop" << std::endl;
  else if(CGAL::assign(sf,o))
    std::cout << "Locating sface" << std::endl;
  else {
    std::cout << "something wrong" << std::endl;
    return 1;
  }
  return 0;
}
```

File: examples/Nef_S2/nef_s2_point_location.cpp

21.3.5 Visualization

Nef_polyhedron_S2 provides an interface for OpenGL visualization via a Qt widget. The usage is shown in the following example:

```
#include <CGAL/basic.h>

#include <CGAL/Gmpz.h>
#include <CGAL/Homogeneous.h>
#include <CGAL/Nef_polyhedron_S2.h>
#include <CGAL/Nef_S2/create_random_Nef_S2.h>
#include <CGAL/IO/Qt_widget_Nef_S2.h>
```

```
#include <qapplication.h>

typedef CGAL::Gmpz RT;
typedef CGAL::Homogeneous<RT> Kernel;
typedef CGAL::Nef_polyhedron_S2<Kernel> Nef_polyhedron_S2;

int main(int argc, char* argv[]) {

  Nef_polyhedron_S2 S;
  create_random_Nef_S2(S,5);

  QApplication a(argc, argv);
  CGAL::Qt_widget_Nef_S2<Nef_polyhedron_S2>* w =
    new CGAL::Qt_widget_Nef_S2<Nef_polyhedron_S2>(S);
  a.setMainWidget(w);
  w->show();
  return a.exec();
}
```

File: demo/Nef_S2/nef_S2.cpp

2D Boolean Operations on Nef Polygons Embedded on the Sphere Reference Manual

Peter Hachenberger, Lutz Kettner, and Michael Seel

Nef polyhedra are defined as a subset of the d-dimensional space obtained by a finite number of set complement and set intersection operations on halfspaces.

Due to the fact that all other binary set operations like union, difference and symmetric difference can be reduced to intersection and complement calculations, Nef polyhedra are also closed under those operations. Also, Nef polyhedra are closed under topological unary set operations. Given a Nef polyhedron one can determine its interior, its boundary, and its closure.

Additionally, a d-dimensional Nef polyhedron has the property, that its boundary is a (d-1)-dimensional Nef polyhedron. This property can be used as a way to represent 3-dimensional Nef polyhedra by means of planar Nef polyhedra. This is done by intersecting the neighborhood of a vertex in a 3D Nef polyhedron with an ε-sphere. The result is a planar Nef polyhedron embedded on the sphere.

The intersection of a halfspace going through the center of the ε-sphere, with the ε-sphere, results in a halfsphere which is bounded by a great circle. A binary operation of two halfspheres cuts the great circles into great arcs.

The incidence structure of planar Nef polyhedra can be reused. The items are denoted as $svertex$, $shalfedge$ and $sface$, analogous to their counterparts in *Nef_polyhedron_2*. Additionally, there is the *shalfloop* representing the great circles.

21.4 Classified Reference Pages

21.5 Alphabetical List of Reference Pages

CGAL::Nef_polyhedron_S2<Traits>

Class

Definition

An instance of data type *Nef_polyhedron_S2<Traits>* is a subset of the sphere S_2 that is the result of forming complements and intersections starting from a finite set H of halfspaces bounded by a plane containing the origin. Halfspaces correspond to hemispheres of S_2 and are therefore modeled by oriented great circles of type *Sphere_circle*. *Nef_polyhedron_S2* is closed under all binary set operations *intersection*, *union*, *difference*, *complement* and under the topological operations *boundary*, *closure*, and *interior*.

#include <CGAL/Nef_polyhedron_S2.h>

Parameters

template < class Nef_polyhedronTraits_S2,
class Nef_polyhedronItems_S2 = CGAL::SM_items,
class Nef_polyhedronMarks = bool
class Nef_polyhedron_S2;

The first parameter requires one of the following exact kernels: *Homogeneous*, *Simple_homogeneous* parametrized with *Gmpz*, *leda_integer* or any other number type modeling $\mathbb{Z}$, or *Cartesian*, *Simple_cartesian* parametrized with *Gmpq*, *leda_rational*,*Quotient<Gmpz>* or any other number type modeling $\mathbb{Q}$.

The second parameter and the third parameter are for future considerations. Neither *Nef_polyhedronItems_S2* nor *Nef_polyhedronMarks* is specifed, yet. Do not use other than the default types for these two template parameters.

Types

Nef_polyhedron_S2<Traits>:: Sphere_point	points in the sphere surface.
Nef_polyhedron_S2<Traits>:: Sphere_segment	segments in the sphere surface.
Nef_polyhedron_S2<Traits>:: Sphere_circle	oriented great circles modeling spatial half-spaces.
Nef_polyhedron_S2<Traits>:: SVertex_const_handle	non-mutable handle to svertex.
Nef_polyhedron_S2<Traits>:: SHalfedge_const_handle	non-mutable handle to shalfedge.
Nef_polyhedron_S2<Traits>:: SHalfloop_const_handle	non-mutable handle to shalfloop.
Nef_polyhedron_S2<Traits>:: SFace_const_handle	non-mutable handle to sface.
Nef_polyhedron_S2<Traits>:: SVertex_const_iterator	non-mutable iterator over all svertices.
Nef_polyhedron_S2<Traits>:: SHalfedge_const_iterator	non-mutable iterator over all shalfedges.
Nef_polyhedron_S2<Traits>:: SHalfloop_const_iterator	non-mutable iterator over all shalfloops.
Nef_polyhedron_S2<Traits>:: SFace_const_iterator	non-mutable iterator over all sfaces.

Nef_polyhedron_S2<Traits>:: SHalfedge_around_svertex_const_circulator

circulating the adjacency list of an svertex v.

Nef_polyhedron_S2<Traits>:: SHalfedge_around_sface_const_circulator

circulating the sface cycle of an sface f.

Nef_polyhedron_S2<Traits>:: SFace_cycle_const_iterator — iterating all sface cycles of an sface f. The iterator has method *bool is_svertex()*, *bool is_shalfedge()*, *bool is_shalfloop()*, and can be converted to the corresponding handles *SVertex_const_handle*, *SHalfedge_const_handle*, or *SHalfloop_const_handle*.

Nef_polyhedron_S2<Traits>:: Mark — attributes of objects (vertices, edges, faces).

Nef_polyhedron_S2<Traits>:: size_type — size type

enum Boundary { *EXCLUDED, INCLUDED*}; — construction selection.

enum Content { *EMPTY, COMPLETE*}; — construction selection.

Creation

Nef_polyhedron_S2<Traits> N(Content sphere = EMPTY);

creates an instance *N* of type *Nef_polyhedron_S2<K>* and initializes it to the empty set if *sphere* == *EMPTY* and to the whole sphere if *sphere* == *COMPLETE*.

Nef_polyhedron_S2<Traits> N(Sphere_circle c, Boundary circle = INCLUDED);

creates a Nef polyhedron *N* containing the half-sphere left of *c* including *c* if *circle==INCLUDED*, excluding *c* if *circle==EXCLUDED*.

template <class Forward_iterator>
Nef_polyhedron_S2<Traits> N(Forward_iterator first, Forward_iterator beyond, Boundary b = INCLUDED)

creates a Nef polyhedron *N* from the set of sphere segments in the iterator range *[first,beyond)*. If the set of sphere segments is a simple polygon that separates the sphere surface into two regions, then the polygonal region that is left of the segment **first* is selected. The polygonal region includes its boundary if *b = INCLUDED* and excludes the boundary otherwise. *Forward_iterator* has to be an iterator with value type *Sphere_segment*.

Operations

void *N.clear(Content plane = EMPTY)*

makes *N* the empty set if *plane* == *EMPTY* and the full plane if *plane* == *COMPLETE*.

bool *N.is_empty()* returns true if *N* is empty, false otherwise.

bool *N.is_sphere()* returns true if *N* is the whole sphere, false otherwise.

Constructive Operations

Nef_polyhedron_S2<K> *N.complement()* returns the complement of *N* in the plane.

Nef_polyhedron_S2<K> *N.interior()* returns the interior of *N*.

Nef_polyhedron_S2<K> *N.closure()* returns the closure of *N*.

Nef_polyhedron_S2<K> *N.boundary()* returns the boundary of *N*.

Nef_polyhedron_S2<K> *N.regularization()* returns the regularized polyhedron (closure of interior).

Nef_polyhedron_S2<K> *N.intersection(Nef_polyhedron_S2<K> N1)*

returns $N \cap N1$.

Nef_polyhedron_S2<K> *N.join(Nef_polyhedron_S2<K> N1)*

returns $N \cup N1$.

Nef_polyhedron_S2<K> *N.difference(Nef_polyhedron_S2<K> N1)*

returns $N - N1$.

Nef_polyhedron_S2<K> *N.symmetric_difference(Nef_polyhedron_S2<K> N1)*

returns the symmectric difference *N - T* $\cup$ *T - N*.

Additionally there are operators **,+,-, ^,!* which implement the binary operations *intersection*, *union*, *difference*, *symmetric difference*, and the unary operation *complement* respectively. There are also the corresponding modification operations **=,+=,-=,^=*.

There are also comparison operations like *<,<=,>,>=,==,!=* which implement the relations subset, subset or equal, superset, superset or equal, equality, inequality, respectively.

Statistics and Integrity

Size_type	*N.number_of_svertices()*	returns the number of svertices.
Size_type	*N.number_of_shalfedges()*	returns the number of shalfedges.
Size_type	*N.number_of_sedges()*	returns the number of sedges.
Size_type	*N.number_of_shalfloops()*	returns the number of shalfloops.
Size_type	*N.number_of_sloops()*	returns the number of sloops.

Size_type	*N.number_of_sfaces()*	returns the number of sfaces.
Size_type	*N.number_of_sface_cycles()*	returns the number of sface cycles.
Size_type	*N.number_of_connected_components()*	calculates the number of connected components of P.
void	*N.print_statistics(std::ostream& os = std::cout)*	print the statistics of P: the number of vertices, edges, and faces.
void	*N.check_integrity_and_topological_planarity(bool faces=true)*	checks the link structure and the genus of P.

Exploration - Point location - Ray shooting

As Nef polyhedra are the result of forming complements and intersections starting from a set H of half-spaces that are defined by oriented lines in the plane, they can be represented by an attributed plane map $M = (V, E, F)$. For topological queries within M the following types and operations allow exploration access to this structure.

Types

Nef_polyhedron_S2<Traits>:: Object_handle

a generic handle to an object of the underlying plane map. The kind of object *(vertex, halfedge, face)* can be determined and the object can be assigned to a corresponding handle by the three functions:
bool assign(Vertex_const_handle& h, Object_handle)
bool assign(Halfedge_const_handle& h, Object_handle)
bool assign(Face_const_handle& h, Object_handle)
where each function returns *true* iff the assignment to h was done.

Operations

bool *N.contains(Object_handle h)*

returns true iff the object h is contained in the set represented by N.

bool *N.contained_in_boundary(Object_handle h)*

returns true iff the object h is contained in the 1-skeleton of N.

Object_handle *N.locate(Sphere_point p)*

returns a generic handle h to an object (face, halfedge, vertex) of the underlying plane map that contains the point p in its relative interior. The point p is contained in the set represented by N if *N.contains(h)* is true. The location mode flag m allows one to choose between different point location strategies.

Object_handle *N.ray_shoot(Sphere_point p, Sphere_direction d)*

returns a handle h with *N.contains(h)* that can be converted to a *Vertex_/Halfedge_/Face_const_handle* as described above. The object returned is intersected by the ray starting in p with direction d and has minimal distance to p. The operation returns an empty *Object_handle* if the ray shoot along d does not hit any object h of N with *N.contains(h)*.

Object_handle *N.ray_shoot_to_boundary(Sphere_point p, Sphere_direction d)*

returns a handle h that can be converted to a *Vertex_/Halfedge_const_handle* as described above. The object returned is part of the 1-skeleton of N, intersected by the ray starting in p with direction d and has minimal distance to p. The operation returns an empty *Object_handle* if the ray shoot along d does not hit any 1-skeleton object h of N. The location mode flag m allows one to choose between different point location strategies.

Iteration

bool *N.has_shalfloop() const*

returns true iff there is a shalfloop.

SHalfloop_const_handle *N.shalfloop() const* returns access to the sloop.

The list of all objects can be accessed via iterator ranges. For comfortable iteration we also provide iterations macros. The iterator range access operations are of the following kind:
SVertex_iterator svertices_begin()/svertices_end()
SHalfedge_iterator shalfedges_begin()/shalfedges_end()
SHalfloop_iterator shalfloops_begin()/shalfloops_end()
SFace_iterator sfaces_begin()/sfaces_end()

The macros are then *CGAL_forall_svertices(v,M)*, *CGAL_forall_shalfedges(e,M)*, *CGAL_forall_sfaces(f,M)*, *CGAL_forall_sface_cycles_of(fc,F)* where M is a sphere map and F is a sface.

Input and Output

A Nef polyhedron N can be visualized in an open GL window. The output operator is defined in the file *CGAL/IO/Nef_polyhedron_2_Window_stream.h*.

Implementation

Nef polyhedra are implemented on top of a halfedge data structure and use linear space in the number of vertices, edges and facets. Operations like *empty* take constant time. The operations *clear*, *complement*, *interior*, *closure*, *boundary*, *regularization*, input and output take linear time. All binary set operations and comparison operations take time $O(n \log n)$ where n is the size of the output plus the size of the input.

The point location and ray shooting operations are implemented in the naive way. The operations run in linear query time without any preprocessing.

Class

CGAL::Nef_polyhedron_S2<Traits>::Sphere_point

Definition

An object p of type *Sphere_point<R>* is a point on the surface of a unit sphere. Such points correspond to the nontrivial directions in space and similarly to the equivalence classes of all nontrivial vectors under normalization.

Types

Sphere_point:: RT	ring number type.

Creation

Sphere_point p;	creates some sphere point.
Sphere_point p(RT x, RT y, RT z);	creates a sphere point corresponding to the point of intersection of the ray starting at the origin in direction (x,y,z) and the surface of S_2.

Operations

Access to the coordinates is provided by the following operations. Note that the vector (x,y,z) is not normalized.

RT	*p.x()*	the x-coordinate.
RT	*p.y()*	the y-coordinate.
RT	*p.z()*	the z-coordinate.
bool	$p == q$	Equality.
bool	$p\ != q$	Inequality.
Sphere_point	*p.antipode()*	returns the antipode of p.

CGAL::Nef_polyhedron_S2<Traits>::Sphere_segment

Class

Definition

An object s of type *Sphere_segment* is a segment in the surface of a unit sphere that is part of a great circle trough the origin. Sphere segments are represented by two sphere points p and q plus an oriented plane h that contains p and q. The plane determines the sphere segment as follows. Let c be the circle in the intersection of h and S_2. Then s is that part of c that is swept, when we rotate p into q in counterclockwise rotation around the normal vector of h as seen from the positive halfspace.

Creation

Sphere_segment s; — creates some sphere segment.

Sphere_segment s(Sphere_point p1, Sphere_point p2, bool shorter_arc=true);

creates a spherical segment spanning the shorter arc from *p1* to *p2* if *shorter_arc* == *true*. Otherwise the longer arc is created.
Precondition: *p1 != p2* and *p1 != p2.opposite()*.

Sphere_segment s(Sphere_point p1, Sphere_point p2, Sphere_circle c);

creates a spherical segment spanning the arc from *p1* to *p2* as part of the oriented circle *c* (*p1 == p2* or *p1 == p2.opposite()* are possible.)
Precondition: *p1* and *p2* are contained in *c*.

Sphere_segment s(Sphere_circle c1, Sphere_circle c2);

creates the spherical segment as part of *c1* that is part of the halfsphere left of the oriented circle *c2*.
Precondition: *c1 != c2* as unoriented circles.

Operations

Sphere_point	*s.source()*	the source point of *s*.
Sphere_point	*s.target()*	the target point of *s*.
Sphere_circle	*s.sphere_circle()*	the great circle supporting *s*.
Sphere_segment	*s.opposite()*	returns the sperical segment oriented from *target()* to *source()* with the same point set as *s*.
Sphere_segment	*s.complement()*	returns the sperical segment oriented from *target()* to *source()* with the point set completing *s* to a full circle.

bool	*s.is_short()*	a segment is short iff it is shorter than a half-circle.
bool	*s.is_long()*	a segment is long iff it is longer than a half-circle.
bool	*s.is_degenerate()*	return true iff *s* is degenerate, i.e. source and target are the same.
bool	*s.is_halfcircle()*	return true iff *s* is a perfect half-circle, i.e. *source().antipode* == *target()*.
bool	*s.has_on(Sphere_point p)*	
		return true iff *s* contains *p*.
bool	*s.has_in_relative_interior(Sphere_point p)*	
		return true iff *s* contains *p* in its relative interior.

CGAL::Nef_polyhedron_S2<Traits>::Sphere_circle

Class

Definition

An object c of type *Sphere_circle* is an oriented great circle on the surface of a unit sphere. Such circles correspond to the intersection of an oriented plane (that contains the origin) and the surface of S_2. The orientation of the great circle is that of a counterclockwise walk along the circle as seen from the positive halfspace of the oriented plane.

Types

Sphere_circle:: RT — ring type.

Sphere_circle:: Plane_3 — plane a *Sphere_circle* lies in.

Creation

Sphere_circle c; — creates some great circle.

Sphere_circle c(Sphere_point p, Sphere_point q);

If p and q are opposite of each other, then we create the unique great circle on S_2 which contains p and q. This circle is oriented such that a walk along c meets p just before the shorter segment between p and q. If p and q are opposite of each other then we create any great circle that contains p and q.

Sphere_circle c(Plane_3 h); — creates the circle corresponding to the plane h.
Precondition: h contains the origin.

Sphere_circle c(RT x, RT y, RT z); — creates the circle orthogonal to the vector (x,y,z).

Sphere_circle c(Sphere_circle c, Sphere_point p);

creates a great circle orthogonal to c that contains p.
Precondition: p is not part of c.

Operations

Sphere_circle	*c.opposite()*	Returns a sphere circle in the oppostie direction of c.
bool	*c.has_on(Sphere_point p)*	returns true iff c contains p.
Plane_3	*c.plane()*	returns the plane supporting c.
Sphere_point	*c.orthogonal_pole()*	returns the point that is the pole of the hemisphere left of c.

Global functions

bool *equal_as_sets(const c1, const c2)*

returns true iff *c1* and *c2* are equal as unoriented circles.

CGAL::Nef_polyhedron_S2<Traits>::SVertex

Class

Definition

Figure 21.2 illustrates the incidence of a svertex on a sphere map.

The member function *out_sedge* returns the first outgoing shalfedge, and *incident_sface* returns the incident sface.

#include <CGAL/Nef_polyhedron_S2.h>

Types

The following types are the same as in *Nef_polyhedron_S2<Traits>*.

SVertex:: Mark	type of mark.
SVertex:: Sphere_point	sphere point type stored in SVertex.
SVertex:: SVertex_const_handle	const handle to SVertex.
SVertex:: SHalfedge_const_handle	const handle to SHalfedge.
SVertex:: SFace_const_handle	const handle to SFace.

Creation

There is no need for a user to create a *SVertex* explicitly. The class *Nef_polyhedron_S2<Traits>* manages the needed svertices internally.

Operations

Mark	*e.mark() const*	the mark of *e* .
Sphere_point	*e.point() const*	the sphere point of *e* .
bool	*e.is_isolated() const*	returns —true— if *e* has no adjacent sedges.
SVertex_const_handle	*e.twin() const*	the twin of *e* .
SHalfedge_const_handle	*e.out_sedge() const*	the first out sedge of *e* .
SFace_const_handle	*e.incident_sface() const*	the incident sface of *e* .

See Also

CGAL::Nef_polyhedron_S2<Traits>::SHalfedge . . . . page 1615
CGAL::Nef_polyhedron_S2<Traits>::SFace . . . . page 1619
CGAL::Nef_polyhedron_S2<Traits>::Sphere_point . . . . page 1608

CGAL::Nef_polyhedron_S2<Traits>::SHalfedge

Definition

A shalfedge is a great arc on a sphere map. The figure below depicts the relationship between a shalfedge and its incident shalfedges, svertices, and sfaces on a sphere map. A shalfedge is an oriented sedge between two svertices. It is always paired with a shalfedge pointing in the opposite direction. The *twin()* member function returns this shalfedge of opposite orientation.

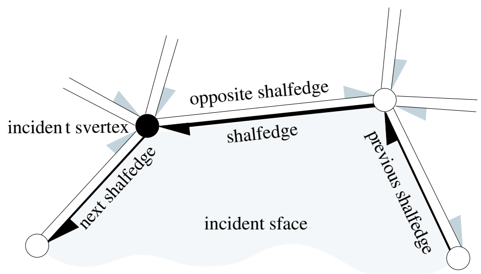


Figure 21.2: Incidences of an SHalfedge

The *snext()* member function points to the successor shalfedge around this sface while the *sprev()* member function points to the preceding shalfedge. An successive assignments of the form *se = se->snext()* cycles counterclockwise around the sface (or hole).

Similarly, the successive assignments of the form *se = se->snext()->twin()* cycle clockwise around the svertex and traverse all halfedges incident to this svertex. The assignment *se = se->cyclic_adj_succ()* can be used as a shortcut.

A const circulator is provided for each of the two circular orders. The circulators are bidirectional and assignable to *SHalfedge_const_handle*.

#include <CGAL/Nef_polyhedron_S2.h>

Types

The following types are the same as in *Nef_polyhedron_S2<Traits>*.

SHalfedge:: Mark	type of mark.
SHalfedge:: Sphere_circle	sphere circle type stored in SHalfedge.
SHalfedge:: SVertex_const_handle	const handle to SVertex.
SHalfedge:: SHalfedge_const_handle	const handle to SHalfedge.
SHalfedge:: SFace_const_handle	const handle to SFace.

Creation

There is no need for a user to create a *SHalfedge* explicitly. The class *Nef_polyhedron_S2<Traits>* manages the needed shalfedges internally.

Operations

Mark	*se.mark() const*	the mark of *se* .
Sphere_circle	*se.circle() const*	the sphere circle of *se* .
SHalfedge_const_handle	*se.twin() const*	the twin of *se* .
SVertex_const_handle	*se.source() const*	the source svertex of *se* .
SVertex_const_handle	*se.target() const*	equals *twin()->source()*.
SHalfedge_const_handle	*se.sprev() const*	the SHalfedge previous to *se* in a sface cycle.
SHalfedge_const_handle	*se.snext() const*	the next SHalfedge of *se* in a sface cycle.
SHalfedge_const_handle	*se.cyclic_adj_pred() const*	the edge before *se* in the cyclic ordered adjacency list of source().
SHalfedge_const_handle	*se.cyclic_adj_succ() const*	the edge after *se* in the cyclic ordered adjacency list of source().
SFace_const_handle	*se.incident_sface() const*	the incident sface of *se* .
bool	*se.in_outer_sface_cycle() const*	determines whether *se* is in an outer sface cycle.
bool	*se.in_inner_sface_cycle() const*	determines whether *se* is in an inner sface cycle.

See Also

CGAL::Nef_polyhedron_S2<Traits>::SVertex . . . . page 1613
CGAL::Nef_polyhedron_S2<Traits>::SFace . . . . page 1619
CGAL::Nef_polyhedron_S2<Traits>::Sphere_circle . . . . page 1611

CGAL::Nef_polyhedron_S2<Traits>::SHalfloop

Definition

A sloop is a great circle on a sphere. A shalfloop is an oriented sloop. It is always paired with a shalfloop whose supporting *Sphere_circle* is pointing in the opposite direction. The *twin()* member function returns this shalfloop of opposite orientation. Each *Nef_polyhedron_S2* can only have one sloop (resp. two shalfloops).

The figure below depicts the relationship between a shalfloop and sfaces on a sphere map.

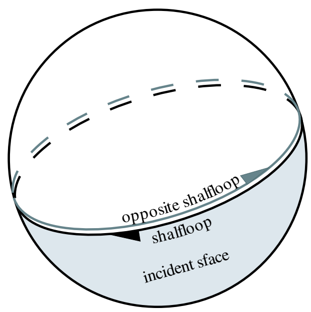


Figure 21.3: Incidences of an SHalfloop

#include <CGAL/Nef_polyhedron_S2.h>

Types

The following types are the same as in *Nef_polyhedron_S2<Traits>*.

SHalfloop:: Mark	type of mark.
SHalfloop:: Sphere_circle	sphere circle type stored in SHalfloop.
SHalfloop:: SHalfloop_const_handle	const handle to SHalfloop.
SHalfloop:: SFace_const_handle	const handle to SFace.

Creation

There is no need for a user to create a *SHalfloop* explicitly. The class *Nef_polyhedron_S2<Traits>* manages the needed shalfloops internally.

Operations

Mark	*se.mark() const*	the mark of *se* .
Sphere_circle	*se.circle() const*	the sphere circle of *se* .

SHalfloop_const_handle	*se.twin() const*	the twin of *se* .
SFace_const_handle	*se.incident_sface() const*	the incident sface of *se* .

See Also

CGAL::Nef_polyhedron_S2<Traits>::SFace . . . . page 1619
CGAL::Nef_polyhedron_S2<Traits>::Sphere_circle . . . . page 1611

CGAL::Nef_polyhedron_S2<Traits>::SFace

Definition

Figures 21.2 and 21.3 illustrate the incidences of an sface. An sface is described by its boundaries. An entry item to each boundary cycle can be accessed using the iterator range (*sface_cycles_begin()*/*sface_cycles_end()*). Additionally, *Nef_polyhedron_S2* provides the macro *CGAL_forall_sface_cylces_of*. The iterators are of type *SFace_cycle_const_iterator* and represent either a shalfedge, a shalfloop, or a svertex.

#include <CGAL/Nef_polyhedron_S2.h>

Types

The following types are the same as in *Nef_polyhedron_S2<Traits>*.

SFace:: Mark	type of mark.
SFace:: Object_list	list of Object handles.
SFace:: Vertex_const_handle	const handle to Vertex.
SFace:: Volume_const_handle	const handle to Volume.
SFace:: SFace_const_handle	const handle to SFace.
SFace:: SFace_cycle_const_iterator	const iterator over the entries to all sface cycles of a sface.

Creation

There is no need for a user to create a *SFace* explicitly. The class *Nef_polyhedron_S2<Traits>* manages the needed sfaces internally.

Operations

Mark	*sf.mark() const*	the mark of *sf*.
SFace_cycle_const_iterator	*sf.sface_cycle_begin() const*	iterator over the entries to all sface cycles of *sf*.
SFace_cycle_const_iterator	*sf.sface_cycle_end() const*	past-the-end iterator.

See Also

CGAL::Nef_polyhedron_S2<Traits> . . . . page 1603
CGAL::Nef_polyhedron_S2<Traits>::SVertex . . . . page 1613

Class

CGAL::Nef_polyhedron_S2<Traits>::SFace_cycle_iterator

Definition

The type *SFace_cycle_iterator* iterates over a list of *Object_handles*. Each item of that list can either be assigned to *SVertex_handle*, *SHalfedge_handle* or *SHalfloop_handle*. To find out which of these assignment works out, the member functions *is_svertex()*, *is_shalfedge()* and *is_shalfloop()* are provided.

#include <CGAL/Nef_polyhedron_S2.h>

Types

SFace_cycle_iterator:: SVertex_handle	const handle to SVertex.
SFace_cycle_iterator:: SHalfedge_handle	const handle to SHalfedge.
SFace_cycle_iterator:: SHalfloop_handle	const handle to SHalfloop.

Creation

SFace_cycle_iterator sfc;	default constructor.

Operations

bool	*sfc.is_svertex() const*	returns true if *sfc* represents a *SVertex_handle*.
bool	*sfc.is_shalfedge() const*	returns true if *sfc* represents a *SHalfedge_handle*.
bool	*sfc.is_shalfloop() const*	returns true if *sfc* represents a *SHalfloop_handle*.
SVertex_handle	*SVertex_handle(sfc)*	casts *sfc* to *SVertex_handle*.
SHalfedge_handle	*SHalfedge_handle(sfc)*	casts *sfc* to *SHalfedge_handle*.
SHalfloop_handle	*SHalfloop_handle(sfc)*	casts *sfc* to *SHalfloop_handle*.

See Also

CGAL::Nef_polyhedron_S2<Traits>::SVertex .... page 1613
CGAL::Nef_polyhedron_S2<Traits>::SHalfedge .... page 1615
CGAL::Nef_polyhedron_S2<Traits>::SHalfloop .... page 1617

CGAL::Qt_widget_Nef_S2<Nef_polyhedron_S2>

Definition

The class *Qt_widget_Nef_S2* uses the OpenGL interface of Qt in order to display a *Nef_polyhedron_S2*. Its purpose is to provide an easy to use viewer for *Nef_polyhedron_S2*. There are no means provided to enhance the functionality of the viewer.

In addition to the functions inherited from the Qt class *QGLWidget*, *Qt_widget_Nef_S2* only has a single public constructor. For the usage of *Qt_widget_Nef_S2* see the example below.

#include <CGAL/IO/Qt_widget_Nef_S2.h>

Parameters

The template parameter expects an instantiation of *Nef_polyhedron_S2<Traits>*.

Creation

Qt_widget_Nef_S2<Nef_polyhedron_S2> W(Nef_polyhedron_S2 N);

Creates a widget *W* for displaying the *Nef_polyhedron_S2* N.

See Also

CGAL::Nef_polyhedron_S2<Traits> . . . . page 1603

Example

This example creates some random *Sphere_segments* and distributes them on two *Nef_polyhedron_2*. The two Nef polyhedra are combined by a symmetric diffrence and the result is displayed in a Qt widget.

```
#include <CGAL/basic.h>

#include <CGAL/Gmpz.h>
#include <CGAL/Homogeneous.h>
#include <CGAL/Nef_polyhedron_S2.h>
#include <CGAL/Nef_S2/create_random_Nef_S2.h>
#include <CGAL/IO/Qt_widget_Nef_S2.h>
#include <qapplication.h>

typedef CGAL::Gmpz RT;
typedef CGAL::Homogeneous<RT> Kernel;
typedef CGAL::Nef_polyhedron_S2<Kernel> Nef_polyhedron_S2;

int main(int argc, char* argv[]) {
```

```
  Nef_polyhedron_S2 S;
  create_random_Nef_S2(S,5);

  QApplication a(argc, argv);
  CGAL::Qt_widget_Nef_S2<Nef_polyhedron_S2>* w =
    new CGAL::Qt_widget_Nef_S2<Nef_polyhedron_S2>(S);
  a.setMainWidget(w);
  w->show();
  return a.exec();
}
```

File: demo/Nef_S2/nef_S2.cpp

Chapter 22

2D Polygon Partitioning

Susan Hert

Contents

22.1 Introduction

A *partition* of a polygon P is a set of polygons such that the interiors of the polygons do not intersect and the union of the polygons is equal to the interior of the original polygon P. This chapter describes functions for partitioning planar polygons into two types of subpolygons — y-monotone polygons and convex polygons. The partitions are produced without introducing new (Steiner) vertices.

All the partitioning functions present the same interface to the user. That is, the user provides a pair of input iterators, *first* and *beyond*, an output iterator *result*, and a traits class *traits*. The points in the range [*first*, *beyond*) are assumed to define a simple polygon whose vertices are in counterclockwise order. The computed partition polygons, whose vertices are also oriented counterclockwise, are written to the sequence starting at position *result* and the past-the-end iterator for the resulting sequence of polygons is returned. The traits classes for the functions specify the types of the input points and output polygons as well as a few other types and function objects that are required by the various algorithms.

22.2 Monotone Partitioning

A *y-monotone polygon* is a polygon whose vertices $v_1,\ldots,v_n$ can be divided into two chains $v_1,\ldots,v_k$ and $v_k,\ldots,v_n,v_1$, such that any horizontal line intersects either chain at most once. For producing a y-monotone partition of a given polygon, the sweep-line algorithm presented in [dBvKOS97] is implemented by the function *y_monotone_partition_2*. This algorithm runs in $O(n \log n)$ time and requires $O(n)$ space. This algorithm does not guarantee a bound on the number of polygons produced with respect to the optimal number.

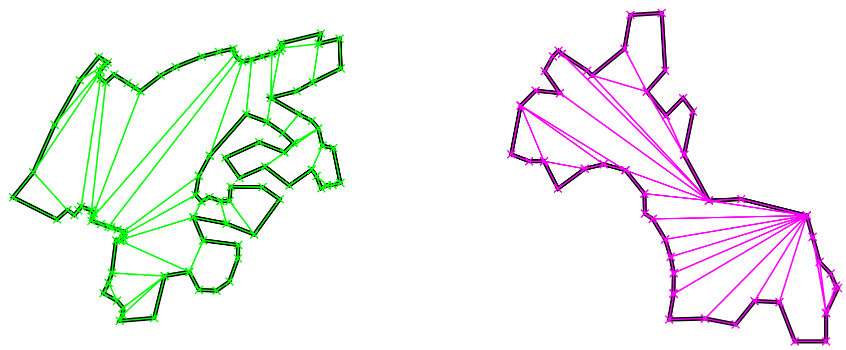


Figure 22.1: Examples of an optimal convex partition (left) and an approximately optimal convex partition (right).

For checking the validity of the partitions produced by *y_monotone_partition_2*, we provide a function *is_y_monotone_2*, which determines if a sequence of points in 2D defines a *y*-monotone polygon or not. For examples of the use of these functions, see the corresponding reference pages.

22.3 Convex Partitioning

Three functions are provided for producing convex partitions of polygons. One produces a partition that is optimal in the number of pieces. The other two functions produce approximately optimal convex partitions. Both these functions produce convex decompositions by first decomposing the polygon into simpler polygons; the first uses a triangulation and the second a monotone partition. These two functions both guarantee that they will produce no more than four times the optimal number of convex pieces but they differ in their runtime complexities. Though the triangulation-based approximation algorithm often results in fewer convex pieces, this is not always the case.

An optimal convex partition can be produced using the function *optimal_convex_partition_2*. This function provides an implementation of Greene's dynamic programming algorithm for optimal partitioning [Gre83]. This algorithm requires $O(n^4)$ time and $O(n^3)$ space in the worst case.

The function *approx_convex_partition_2* implements the simple approximation algorithm of Hertel and Mehlhorn [HM83] that produces a convex partitioning of a polygon from a triangulation by throwing out unnecessary triangulation edges. The triangulation used in this function is one produced by the 2-dimensional constrained triangulation package of CGAL. For a given triangulation, this convex partitioning algorithm requires $O(n)$ time and space to construct a decomposition into no more than four times the optimal number of convex pieces.

The sweep-line approximation algorithm of Greene [Gre83], which, given a monotone partition of a polygon, produces a convex partition in $O(n \log n)$ time and $O(n)$ space, is implemented by the function *greene_approx_convex_partition_2*. The function *y_monotone_partition_2* described in Section 22.2 is used to produce the monotone partition. This algorithm provides the same worst-case approximation guarantee as the algorithm of Hertel and Mehlhorn implemented with *approx_convex_partition_2* but can sometimes produce better results (*i.e.*, convex partitions with fewer pieces).

Examples of the uses of all of these functions are provided with the corresponding reference pages.

2D Polygon Partitioning Reference Manual

Susan Hert

A *partition* of a polygon is a set of polygons such that the interiors of the polygons do not intersect and the union of the polygons is equal to the interior of the original polygon. Functions are available for partitioning planar polygons into two types of subpolygons — y-monotone polygons and convex polygons.

The function that produces a y-monotone partitioning is based on the algorithm presented in [dBvKOS97] which requires $O(n \log n)$ time and $O(n)$ space for a polygon with n vertices and guarantees nothing about the number of polygons produced with respect to the optimal number. Three functions are provided for producing convex partitions. Two of these functions produce approximately optimal partitions and one results in an optimal partition, where "optimal" is defined in terms of the number of partition polygons. The two functions that implement approximation algorithms are guaranteed to produce no more than four times the optimal number of convex pieces. The optimal partitioning function provides an implementation of Greene's dynamic programming algorithm [Gre83], which requires $O(n^4)$ time and $O(n^3)$ space to produce a convex partitioning. One of the approximation algorithms is also due to Greene [Gre83] and requires $O(n \log n)$ time and $O(n)$ space to produce a convex partitioning given a y-monotone partitioning. The other approximation algorithm is a result of Hertel and Mehlhorn [HM83], which requires $O(n)$ time and space to produce a convex partitioning from a triangulation of a polygon. Each of the partitioning functions uses a traits class to supply the primitive types and predicates used by the algorithms.

Assertions

The assertion flags for this package use *PARTITION* in their names (*e.g.*, *CGAL_PARTITION_NO_POSTCONDITIONS*). The precondition checks for the planar polygon partitioning functions are: counterclockwise ordering of the input vertices and simplicity of the polygon these vertices represent. The postcondition checks are: simplicity, counterclockwise orientation, and convexity (or y-monotonicity) of the partition polygons and validity of the partition (*i.e.*, the partition polygons are nonoverlapping and the union of these polygons is the same as the original polygon) .

22.4 Classified Reference Pages

Concepts

Function Object Concepts

Classes

Function Object Classes

Functions

22.5 Alphabetical List of Reference Pages

Function

CGAL::approx_convex_partition_2

Definition

Function that produces a set of convex polygons that represent a partitioning of a polygon defined on a sequence of points. The number of convex polygons produced is no more than four times the minimal number.

#include <CGAL/partition_2.h>

template <class InputIterator, class OutputIterator, class Traits>
OutputIterator approx_convex_partition_2(InputIterator first, InputIterator beyond, OutputIterator result, Traits traits = Default_traits)

computes a partition of the polygon defined by the points in the range [*first*, *beyond*) into convex polygons. The counterclockwise-oriented partition polygons are written to the sequence starting at position *result*. The past-the-end iterator for the resulting sequence of polygons is returned.
Precondition: The points in the range [*first*, *beyond*) define a simple counterclockwise-oriented polygon.

Requirements

1. *Traits* is a model of the concept *PartitionTraits_2* and, for the purposes of checking the postcondition that the partition produced is valid, it should also be a model of the concept *ConvexPartitionIsValidTraits_2*.
2. *OutputIterator::value_type* should be *Traits::Polygon_2*.
3. *InputIterator::value_type* should be *Traits::Point_2*, which should also be the type of the points stored in an object of type *Traits::Polygon_2*.
4. Points in the range $[first, beyond)$ must define a simple polygon whose vertices are oriented counterclockwise.

The default traits class *Default_traits* is *Partition_traits_2*, with the representation type determined by *InputIterator1::value_type*.

See Also

CGAL::convex_partition_is_valid_2 . . . . . . page 1633
CGAL::greene_approx_convex_partition_2 . . . . . . page 1636
CGAL::optimal_convex_partition_2 . . . . . . page 1646
CGAL::partition_is_valid_2 . . . . . . page 1651
CGAL::Partition_is_valid_traits_2<Traits, PolygonIsValid> . . . . . . page 1657
CGAL::y_monotone_partition_2 . . . . . . page 1662

Implementation

This function implements the algorithm of Hertel and Mehlhorn [HM83] and is based on the class *CGAL::Constrained_triangulation_2*. Given a triangulation of the polygon, the function requires $O(n)$ time and space for a polygon with n vertices.

Example

The following program computes an approximately optimal convex partitioning of a polygon using the default traits class and stores the partition polygons in the list *partition_polys*.

```
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/Partition_traits_2.h>
#include <CGAL/partition_2.h>
#include <CGAL/point_generators_2.h>
#include <CGAL/random_polygon_2.h>
#include <cassert>
#include <list>


typedef CGAL::Exact_predicates_inexact_constructions_kernel K;
typedef CGAL::Partition_traits_2<K>                         Traits;
typedef Traits::Point_2                                     Point_2;
typedef Traits::Polygon_2                                   Polygon_2;
typedef Polygon_2::Vertex_iterator
   Vertex_iterator;
typedef std::list<Polygon_2>                                Polygon_list;
typedef CGAL::Creator_uniform_2<int, Point_2>               Creator;
typedef CGAL::Random_points_in_square_2<Point_2, Creator>
   Point_generator;

void make_polygon(Polygon_2& polygon)
{
   polygon.push_back(Point_2(391, 374));
   polygon.push_back(Point_2(240, 431));
   polygon.push_back(Point_2(252, 340));
   polygon.push_back(Point_2(374, 320));
   polygon.push_back(Point_2(289, 214));
   polygon.push_back(Point_2(134, 390));
   polygon.push_back(Point_2( 68, 186));
   polygon.push_back(Point_2(154, 259));
   polygon.push_back(Point_2(161, 107));
   polygon.push_back(Point_2(435, 108));
   polygon.push_back(Point_2(208, 148));
   polygon.push_back(Point_2(295, 160));
   polygon.push_back(Point_2(421, 212));
   polygon.push_back(Point_2(441, 303));
}


int main()
{
   Polygon_2    polygon;
   Polygon_list partition_polys;
```

```
/*
   CGAL::random_polygon_2(50, std::back_inserter(polygon),
                          Point_generator(100));
*/
   make_polygon(polygon);
   CGAL::approx_convex_partition_2(polygon.vertices_begin(),
                                   polygon.vertices_end(),
                                   std::back_inserter(partition_polys));
   assert(CGAL::convex_partition_is_valid_2(polygon.vertices_begin(),
                                            polygon.vertices_end(),
                                            partition_polys.begin(),
                                            partition_polys.end()));
   return 0;
}
```

File: examples/Partition_2/approx_convex_partition_2.cpp

CGAL::convex_partition_is_valid_2

Definition

Function that determines if a given set of polygons represents a valid convex partitioning for a given sequence of points that represent a simple, counterclockwise-oriented polygon. A convex partition is valid if the polygons do not overlap, the union of the polygons is the same as the original polygon given by the sequence of points, and if each partition polygon is convex.

#include <CGAL/partition_is_valid_2.h>

template<class InputIterator, class ForwardIterator, class Traits>
bool convex_partition_is_valid_2(InputIterator point_first, InputIterator point_beyond, ForwardIterator poly_first, ForwardIterator poly_beyond, Traits traits = Default_traits)

determines if the polygons in the range [*poly_first*, *poly_beyond*) define a valid convex partition of the polygon defined by the points in the range [*point_first*, *point_beyond*). The function returns *true* iff the partition is valid and otherwise returns *false*.
Precondition: The points in the range [*point_first*, *point_beyond*) define a simple, counterclockwise-oriented polygon.

Requirements

1. *Traits* is a model of the concept *ConvexPartitionIsValidTraits_2*.
2. *InputIterator::value_type* should be *Traits::Point_2*, which should also be the type of the points stored in an object of type *Traits::Polygon_2*.
3. *ForwardIterator::value_type* should be *Traits::Polygon_2*.

The default traits class *Default_traits* is *Partition_traits_2*, with the representation type determined by *InputIterator::value_type*.

See Also

CGAL::approx_convex_partition_2 . . . . page 1630
CGAL::greene_approx_convex_partition_2 . . . . page 1636
CGAL::optimal_convex_partition_2 . . . . page 1646
CGAL::partition_is_valid_2 . . . . page 1651
CGAL::is_convex_2

Implementation

This function calls *partition_is_valid_2* using the function object *Is_convex_2* to determine the convexity of each partition polygon. Thus the time required by this function is $O(n \log n + e \log e)$ where n is the total number of vertices in the partition polygons and e the total number of edges.

Example

See the example presented with the function *approx_convex_partition_2* for an illustration of the use of this function.

advanced

advanced

ConvexPartitionIsValidTraits_2

Concept

Definition

Requirements of a traits class used by *convex_partition_is_valid_2* for testing the validity of a convex partition of a polygon.

Types

All types required by the concept *PartitionIsValidTraits_2* are required except the function object type *Is_valid*. The following type is required instead:

ConvexPartitionIsValidTraits_2:: Is_convex_2 — Model of the concept *PolygonIsValid* that tests if a sequence of points is convex or not.

Creation

Only a copy constructor is required.

ConvexPartitionIsValidTraits_2 traits(& tr);

Operations

The following function that creates an instance of the above predicate object type must exist instead of the function *is_valid_object* required by *PartitionIsValidTraits_2*.

Is_convex_2 *traits.is_convex_2_object(t)*

Has Models

CGAL::Partition_traits_2<R> . . . . page 1659

See Also

CGAL::approx_convex_partition_2 . . . . page 1630
CGAL::greene_approx_convex_partition_2 . . . . page 1636
CGAL::Is_convex_2<Traits> . . . . page 1643
CGAL::optimal_convex_partition_2 . . . . page 1646

advanced

CGAL::greene_approx_convex_partition_2

Definition

Function that produces a set of convex polygons that represent a partitioning of a polygon defined on a sequence of points. The number of convex polygons produced is no more than four times the minimal number.

#include <CGAL/partition_2.h>

template <class InputIterator, class OutputIterator, class Traits>
OutputIterator greene_approx_convex_partition_2(InputIterator first, InputIterator beyond, OutputIterator result, Traits traits = Default_traits)

computes a partition of the polygon defined by the points in the range [*first*, *beyond*) into convex polygons. The counterclockwise-oriented partition polygons are written to the sequence starting at position *result*. The past-the-end iterator for the resulting sequence of polygons is returned.
Precondition: The points in the range [*first*, *beyond*) define a simple, counterclockwise-oriented polygon.

Requirements

1. *Traits* is a model of the concepts *PartitionTraits_2* and *YMonotonePartitionTraits_2*. For the purpose of checking the validity of the y-monotone partition produced as a preprocessing step for the convex partitioning, it must also be a model of *YMonotonePartitionIsValidTraits_2*. For the purpose of checking the postcondition that the convex partition is valid, *Traits* must also be a model of *ConvexPartitionIsValidTraits_2*.
2. *OutputIterator::value_type* is equivalent to *Traits::Polygon_2*.
3. *InputIterator::value_type* is equivalent to *Traits::Point_2*, which should also be equivalent to the type of the points stored in an object of type *Traits::Polygon_2*.

The default traits class *Default_traits* is *Partition_traits_2*, with the representation type determined by *InputIterator::value_type*.

See Also

CGAL::approx_convex_partition_2 . . . . page 1630
CGAL::convex_partition_is_valid_2 . . . . page 1633
CGAL::optimal_convex_partition_2 . . . . page 1646
CGAL::partition_is_valid_2 . . . . page 1651
CGAL::y_monotone_partition_2 . . . . page 1662

Implementation

This function implements the approximation algorithm of Greene [Gre83] and requires $O(n \log n)$ time and $O(n)$ space to produce a convex partitioning given a y-monotone partitioning of a polygon with n vertices. The function *y_monotone_partition_2* is used to produce the monotone partition.

Example

The following program computes an approximately optimal convex partitioning of a polygon using the default traits class and stores the partition polygons in the list *partition_polys*.

```
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/Partition_traits_2.h>
#include <CGAL/partition_2.h>
#include <CGAL/point_generators_2.h>
#include <CGAL/random_polygon_2.h>
#include <cassert>
#include <list>

typedef CGAL::Exact_predicates_inexact_constructions_kernel K;
typedef CGAL::Partition_traits_2<K>                         Traits;
typedef Traits::Point_2                                     Point_2;
typedef Traits::Polygon_2                                   Polygon_2;
typedef Polygon_2::Vertex_iterator
   Vertex_iterator;
typedef std::list<Polygon_2>                                Polygon_list;
typedef CGAL::Creator_uniform_2<int, Point_2>               Creator;
typedef CGAL::Random_points_in_square_2< Point_2, Creator >
   Point_generator;

void make_polygon(Polygon_2& polygon)
{
   polygon.push_back(Point_2(391, 374));
   polygon.push_back(Point_2(240, 431));
   polygon.push_back(Point_2(252, 340));
   polygon.push_back(Point_2(374, 320));
   polygon.push_back(Point_2(289, 214));
   polygon.push_back(Point_2(134, 390));
   polygon.push_back(Point_2( 68, 186));
   polygon.push_back(Point_2(154, 259));
   polygon.push_back(Point_2(161, 107));
   polygon.push_back(Point_2(435, 108));
   polygon.push_back(Point_2(208, 148));
   polygon.push_back(Point_2(295, 160));
   polygon.push_back(Point_2(421, 212));
   polygon.push_back(Point_2(441, 303));
}


int main()
{
   Polygon_2    polygon;
   Polygon_list partition_polys;
   Traits       partition_traits;
```

```
/*
   CGAL::random_polygon_2(50, std::back_inserter(polygon),
                          Point_generator(100));
*/
   make_polygon(polygon);
   CGAL::greene_approx_convex_partition_2(polygon.vertices_begin(),
                                          polygon.vertices_end(),

    std::back_inserter(partition_polys),
                                          partition_traits);
   assert(CGAL::convex_partition_is_valid_2(polygon.vertices_begin(),
                                            polygon.vertices_end(),
                                            partition_polys.begin(),
                                            partition_polys.end(),
                                            partition_traits));
   return 0;
}
```

File: examples/Partition_2/greene_approx_convex_partition_2.cpp

CGAL::is_y_monotone_2

Function

Definition

Function for testing the y-monotonicity of a sequence of points.

#include <CGAL/is_y_monotone_2.h>

template<class InputIterator, class Traits>
bool *is_y_monotone_2(InputIterator first, InputIterator beyond, Traits traits)*

Determines if the sequence of points in the range [*first*, *beyond*) define a y-monotone polygon or not. If so, the function returns *true*, otherwise it returns *false*.

Requirements

1. *Traits* is a model of the concept *IsYMonotoneTraits_2*.
2. *InputIterator::value_type* should be *Traits::Point_2*.

The default traits class *Default_traits* is the kernel in which the type *InputIterator::value_type* is defined.

See Also

CGAL::Is_y_monotone_2<Traits> . . . . page 1645
CGAL::y_monotone_partition_2 . . . . page 1662
CGAL::y_monotone_partition_is_valid_2 . . . . page 1665

Implementation

This function requires $O(n)$ time for a polygon with n vertices.

Example

The following program computes a y-monotone partitioning of a polygon using the default traits class and stores the partition polygons in the list *partition_polys*. It then asserts that each of the partition polygons is, in fact, a y-monotone polygon and that the partition is valid. (Note that the assertions are superfluous unless the postcondition checking done by *y_monotone_partition_2* has been turned off during compilation.)

```
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/Partition_traits_2.h>
#include <CGAL/partition_2.h>
#include <CGAL/point_generators_2.h>
#include <CGAL/random_polygon_2.h>
#include <cassert>
#include <list>
```

```
typedef CGAL::Exact_predicates_inexact_constructions_kernel K;
typedef CGAL::Partition_traits_2<K>                         Traits;
typedef Traits::Point_2                                     Point_2;
typedef Traits::Polygon_2                                   Polygon_2;
typedef std::list<Polygon_2>                                Polygon_list;
typedef CGAL::Creator_uniform_2<int, Point_2>               Creator;
typedef CGAL::Random_points_in_square_2<Point_2, Creator>
   Point_generator;

void make_polygon(Polygon_2& polygon)
{
   polygon.push_back(Point_2(391, 374));
   polygon.push_back(Point_2(240, 431));
   polygon.push_back(Point_2(252, 340));
   polygon.push_back(Point_2(374, 320));
   polygon.push_back(Point_2(289, 214));
   polygon.push_back(Point_2(134, 390));
   polygon.push_back(Point_2( 68, 186));
   polygon.push_back(Point_2(154, 259));
   polygon.push_back(Point_2(161, 107));
   polygon.push_back(Point_2(435, 108));
   polygon.push_back(Point_2(208, 148));
   polygon.push_back(Point_2(295, 160));
   polygon.push_back(Point_2(421, 212));
   polygon.push_back(Point_2(441, 303));
}


int main( )
{
   Polygon_2    polygon;
   Polygon_list partition_polys;

/*
   CGAL::random_polygon_2(50, std::back_inserter(polygon),
                          Point_generator(100));
*/
   make_polygon(polygon);
   CGAL::y_monotone_partition_2(polygon.vertices_begin(),
                                polygon.vertices_end(),
                                std::back_inserter(partition_polys));

   std::list<Polygon_2>::const_iterator   poly_it;
   for (poly_it = partition_polys.begin(); poly_it !=
   partition_polys.end();
        poly_it++)
   {
      assert(CGAL::is_y_monotone_2((*poly_it).vertices_begin(),
                                   (*poly_it).vertices_end()));
   }

   assert(CGAL::partition_is_valid_2(polygon.vertices_begin(),
```

```
                                                    polygon.vertices_end(),
                                                    partition_polys.begin(),
                                                    partition_polys.end()));
   return 0;
}
```

File: examples/Partition_2/y_monotone_partition_2.cpp

Concept

IsYMonotoneTraits_2

Definition

Requirements of a traits class to be used with the function *is_y_monotone_2* that tests whether a sequence of 2D points defines a y-monotone polygon or not.

Types

The following two types are required:

IsYMonotoneTraits_2:: Point_2	The point type of the polygon vertices.
IsYMonotoneTraits_2:: Less_yx_2	Predicate object type that compares *Point_2*s lexicographically. Must provide *bool operator()(Point_2 p, Point_2 q)* where *true* is returned iff $p <_{xy} q$. We have $p <_{xy} q$, iff $p_x < q_x$ or $p_x = q_x$ and $p_y < q_y$, where p_x and p_y denote x and y coordinate of point p resp.

Creation

Only a copy constructor is required.

IsYMonotoneTraits_2 traits(& tr);

Operations

The following function that creates an instance of the above predicate object type must exist:

Less_yx_2	*traits.less_yx_2_object()*

Has Models

CGAL::Partition_traits_2<R> . . . . page 1659
CGAL::Kernel_traits_2

See Also

CGAL::Is_y_monotone_2<Traits> . . . . page 1645
CGAL::y_monotone_partition_2 . . . . page 1662
CGAL::y_monotone_partition_is_valid_2 . . . . page 1665

CGAL::Is_convex_2<Traits>

Functor
Class

Definition

Function object class for testing if a sequence of points represents a convex polygon or not.

#include <CGAL/polygon_function_objects.h>

Is Model for the Concepts

PolygonIsValid . . . . page 1661

Creation

Is_convex_2<Traits> f(Traits t); — *Traits* satisfies the requirements of the function *is_convex_2*

Operations

template<class InputIterator>
bool *f(InputIterator first, InputIterator beyond)*

returns *true* iff the points of type *Triats::Point_2* in the range [*first*,*beyond*) define a convex polygon.

See Also

CGAL::convex_partition_is_valid_2 . . . . page 1633
CGAL::Partition_is_valid_traits_2<Traits, PolygonIsValid> . . . . page 1657

Implementation

This test requires $O(n)$ time for a polygon with n vertices.

Functor
Class

CGAL::Is_vacuously_valid<Traits>

Definition

Function object class that indicates all sequences of points are valid.

#include <CGAL/polygon_function_objects.h>

Is Model for the Concepts

PolygonIsValid . . . . page 1661

Creation

Is_vacuously_valid<Traits> f(Traits t);

Operations

template<class InputIterator>
bool *f(InputIterator first, InputIterator beyond)* returns *true*.

See Also

CGAL::partition_is_valid_2 . . . . page 1651
CGAL::Partition_is_valid_traits_2<Traits, PolygonIsValid> . . . . page 1657

Implementation

This test requires $O(1)$ time.

CGAL::Is_y_monotone_2<Traits>

Functor
Class

Definition

Function object class that tests whether a sequence of points represents a y-monotone polygon or not.

#include <CGAL/polygon_function_objects.h>

Is Model for the Concepts

PolygonIsValid . . . . page 1661

Creation

Is_y_monotone_2<Traits> f(Traits t); *Traits* is a model of the concept *IsYMonotoneTraits_2*

Operations

template<class InputIterator>
bool *f(InputIterator first, InputIterator beyond)*

returns *true* iff the points of type *Traits::Point_2* in the range [*first*,*beyond*) define a y-monotone polygon.

See Also

CGAL::convex_partition_is_valid_2 . . . . page 1633
CGAL::Partition_is_valid_traits_2<Traits, PolygonIsValid> . . . . page 1657

Implementation

This test requires $O(n)$ time for a polygon with n vertices.

Function

CGAL::optimal_convex_partition_2

Function that produces a set of convex polygons that represent a partitioning of a polygon defined on a sequence of points. The number of convex polygons produced is minimal.

#include <CGAL/partition_2.h>

template <class InputIterator, class OutputIterator, class Traits>
OutputIterator optimal_convex_partition_2(InputIterator first, InputIterator beyond, OutputIterator result, Traits traits = Default_traits)

computes a partition of the polygon defined by the points in the range [*first*, *beyond*) into convex polygons. The counterclockwise-oriented partition polygons are written to the sequence starting at position *result*. The past-the-end iterator for the resulting sequence of polygons is returned.
Precondition: The points in the range [*first*, *beyond*) define a simple, counterclockwise-oriented polygon.

Requirements

1. *Traits* is a model of the concept *OptimalConvexPartitionTraits_2*. For the purposes of checking the postcondition that the partition is valid, *Traits* should also be a model of *ConvexPartitionIsValidTraits_2*.

2. *OutputIterator::value_type* should be *Traits::Polygon_2*.

3. *InputIterator::value_type* should be *Traits::Point_2*, which should also be the type of the points stored in an object of type *Traits::Polygon_2*.

The default traits class *Default_traits* is *Partition_traits_2*, with the representation type determined by *InputIterator::value_type*.

See Also

CGAL::approx_convex_partition_2 . . . . page 1630
CGAL::convex_partition_is_valid_2 . . . . page 1633
CGAL::greene_approx_convex_partition_2 . . . . page 1636
CGAL::partition_is_valid_2 . . . . page 1651
CGAL::Partition_is_valid_traits_2<Traits, PolygonIsValid> . . . . page 1657

Implementation

This function implements the dynamic programming algorithm of Greene [Gre83], which requires $O(n^4)$ time and $O(n^3)$ space to produce a partitioning of a polygon with n vertices.

Example

The following program computes an optimal convex partitioning of a polygon using the default traits class and stores the partition polygons in the list *partition_polys*. It then asserts that the partition produced is valid. The traits class used for testing the validity is derived from the traits class used to produce the partition with the function object class *CGAL::Is_convex_2* used to define the required *Is_valid* type. (Note that this assertion is superfluous unless the postcondition checking for *optimal_convex_partition_2* has been turned off.)

```
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/Partition_traits_2.h>
#include <CGAL/Partition_is_valid_traits_2.h>
#include <CGAL/polygon_function_objects.h>
#include <CGAL/partition_2.h>
#include <CGAL/point_generators_2.h>
#include <CGAL/random_polygon_2.h>
#include <cassert>
#include <list>


typedef CGAL::Exact_predicates_inexact_constructions_kernel K;
typedef CGAL::Partition_traits_2<K>                         Traits;
typedef CGAL::Is_convex_2<Traits>                           Is_convex_2;
typedef Traits::Polygon_2                                   Polygon_2;
typedef Traits::Point_2                                     Point_2;
typedef Polygon_2::Vertex_const_iterator
   Vertex_iterator;
typedef std::list<Polygon_2>                                Polygon_list;
typedef CGAL::Partition_is_valid_traits_2<Traits, Is_convex_2>

   Validity_traits;
typedef CGAL::Creator_uniform_2<int, Point_2>               Creator;
typedef CGAL::Random_points_in_square_2<Point_2, Creator>
   Point_generator;

void make_polygon(Polygon_2& polygon)
{
   polygon.push_back(Point_2(391, 374));
   polygon.push_back(Point_2(240, 431));
   polygon.push_back(Point_2(252, 340));
   polygon.push_back(Point_2(374, 320));
   polygon.push_back(Point_2(289, 214));
   polygon.push_back(Point_2(134, 390));
   polygon.push_back(Point_2( 68, 186));
   polygon.push_back(Point_2(154, 259));
   polygon.push_back(Point_2(161, 107));
   polygon.push_back(Point_2(435, 108));
   polygon.push_back(Point_2(208, 148));
   polygon.push_back(Point_2(295, 160));
   polygon.push_back(Point_2(421, 212));
   polygon.push_back(Point_2(441, 303));
}

int main()
{
   Polygon_2             polygon;
```

```
   Polygon_list           partition_polys;
   Traits                 partition_traits;
   Validity_traits        validity_traits;

/*
   CGAL::random_polygon_2(50, std::back_inserter(polygon),
                          Point_generator(100));
*/
   make_polygon(polygon);
   CGAL::optimal_convex_partition_2(polygon.vertices_begin(),
                                    polygon.vertices_end(),
                                    std::back_inserter(partition_polys),
                                    partition_traits);
   assert(CGAL::partition_is_valid_2(polygon.vertices_begin(),
                                     polygon.vertices_end(),
                                     partition_polys.begin(),
                                     partition_polys.end(),
                                     validity_traits));
   return 0;
}
```

File: examples/Partition_2/optimal_convex_partition_2.cpp

OptimalConvexPartitionTraits_2

Concept

Definition

Requirements of a traits class to be used with the function *optimal_convex_partition_2* that computes an optimal convex partition of a polygon.

Refines

PartitionTraits_2 . . . . page 1655

Types

In addition to the types listed with the concept *PartitionTraits_2*, the following types are required:

OptimalConvexPartitionTraits_2:: Segment_2 A segment type

OptimalConvexPartitionTraits_2:: Ray_2 A ray type

OptimalConvexPartitionTraits_2:: Object_2 A general object type that can be either a point or a segment

OptimalConvexPartitionTraits_2:: Construct_segment_2

Function object type that provides *Segment_2 operator()(Point_2 p, Point_2 q)*, which constructs and returns the segment defined by the points p and q.

OptimalConvexPartitionTraits_2:: Construct_ray_2

Function object type that provides *Ray_2 operator()(Point_2 p, Point_2 q)*, which constructs and returns the ray from point p through point q.

OptimalConvexPartitionTraits_2:: Collinear_are_ordered_along_line_2

Predicate object type that determines orderings of *Point_2*s on a line. Must provide *bool operator()(Point_2 p, Point_2 q, Point_2 r)* that returns *true*, iff q lies between p and r and p, q, and r satisfy the precondition that they are collinear.

OptimalConvexPartitionTraits_2:: Are_strictly_ordered_along_line_2

Predicate object type that determines orderings of *Point_2*s. Must provide *bool operator()(Point_2 p, Point_2 q, Point_2 r)* that returns *true*, iff the three points are collinear and q lies strictly between p and r. Note that *false* should be returned if $q{=}{=}p$ or $q{=}{=}r$.

OptimalConvexPartitionTraits_2:: Intersect_2	Function object type that provides *Object_2 operator()(Segment_2 s1, Segment_2 s2)* that returns the intersection of two segments (which may be either a segment or a point).
OptimalConvexPartitionTraits_2:: Assign_2	Function object type that provides *bool operator()(Segment_2 s1, Object_2 o)* that returns *true* if *o* is a segment and assigns the value of *o* to *s1*; returns *false* otherwise.

Creation

Only a copy constructor is required.

OptimalConvexPartitionTraits_2 traits(& tr);

Operations

In addition to the functions required by *PartitionTraits_2*, the following functions that create instances of the above function object types must exist:

Collinear_are_ordered_along_line_2	*traits.collinear_are_ordered_along_line_2_object() const*
Construct_segment_2	*traits.construct_segment_2_object() const*
Construct_ray_2	*traits.construct_ray_2_object() const*
Are_strictly_ordered_along_line_2	*traits.are_strictly_ordered_along_line_2_object() const*

Has Models

CGAL::Partition_traits_2<R> . . . . page 1659

See Also

CGAL::convex_partition_is_valid_2 . . . . page 1633
CGAL::Partition_is_valid_traits_2<Traits, PolygonIsValid> . . . . page 1657

advanced

CGAL::partition_is_valid_2

Definition

Function that determines if a given set of polygons represents a valid partition for a given sequence of points that define a simple, counterclockwise-oriented polygon. A valid partition is one in which the polygons are nonoverlapping and the union of the polygons is the same as the original polygon.

#include <CGAL/partition_is_valid_2.h>

```
template<class InputIterator, class ForwardIterator, class Traits>
bool    partition_is_valid_2( InputIterator point_first,
                              InputIterator point_beyond,
                              ForwardIterator poly_first,
                              ForwardIterator poly_beyond,
                              Traits traits = Default_traits)
```

returns *true* iff the polygons in the range [*poly_first*, *poly_beyond*) define a valid partition of the polygon defined by the points in the range [*point_first*, *point_beyond*) and *false* otherwise. Each polygon must also satisfy the property tested by *Traits::Is_valid()*.
Precondition: Points in the range [*point_first*, *point_beyond*) define a simple, counterclockwise-oriented polygon.

Requirements

1. *Traits* is a model of the concept *PartitionIsValidTraits_2* and the concept defining the requirements for the validity test implemented by *Traits::Is_valid()*.
2. *InputIterator::value_type* should be *Traits::Point_2*, which should also be the type of the points stored in an object of type *Traits::Polygon_2*.
3. *ForwardIterator::value_type* should be *Traits::Polygon_2*.

The default traits class *Default_traits* is *Partition_traits_2*, with the representation type determined by *InputIterator::value_type*.

See Also

CGAL::approx_convex_partition_2 . . . . . . page 1630
CGAL::greene_approx_convex_partition_2 . . . . . . page 1636
CGAL::is_y_monotone_2 . . . . . . page 1639
CGAL::optimal_convex_partition_2 . . . . . . page 1646
CGAL::Partition_is_valid_traits_2<Traits, PolygonIsValid> . . . . . . page 1657
CGAL::y_monotone_partition_2 . . . . . . page 1662
CGAL::is_convex_2

Implementation

This function requires $O(n \log n + e \log e + \Sigma_{i=1}^{p} m_i)$ where n is the total number of vertices of the p partition polygons, e is the total number of edges of the partition polygons and m_i is the time required by *Traits::Is_valid()* to test if partition polygon p_i is valid.

Example

See the example presented with the function *optimal_convex_partition_2* for an illustration of the use of this function.

advanced

advanced

PartitionIsValidTraits_2

Definition

Requirements of a traits class that is used by *partition_is_valid_2*, *convex_partition_is_valid_2*, and *y_monotone_partition_is_valid_2* for testing if a given set of polygons are nonoverlapping and if their union is a polygon that is the same as a polygon represented by a given sequence of points. Note that the traits class for *partition_is_valid_2* may have to satisfy additional requirements if each partition polygon is to be tested for having a particular property; see, for example, the descriptions of the function *is_convex_2* and the concept *YMonotonePartitionTraits_2* for the additional requirements for testing for convexity and y-monotonicity, respectively.

Types

PartitionIsValidTraits_2:: Point_2 — The point type on which the partitioning algorithm operates.

PartitionIsValidTraits_2:: Polygon_2 — The polygon type created by the partitioning function. This type should provide a nested type *Vertex_const_iterator* that is the type of the non-mutable iterator over the polygon vertices.

PartitionIsValidTraits_2:: Is_valid — A model of the concept *PolygonIsValid*

PartitionIsValidTraits_2:: Less_xy_2 — Predicate object type that compares *Point_2*s lexicographically. Must provide *bool operator()(Point_2 p, Point_2 q)* where *true* is returned iff $p <_{xy} q$. We have $p <_{xy} q$, iff $p_x < q_x$ or $p_x = q_x$ and $p_y < q_y$, where p_x and p_y denote the x and y coordinates of point p, respectively.

PartitionIsValidTraits_2:: Left_turn_2 — Predicate object type that provides *bool operator()(Point_2 p,Point_2 q,Point_2 r)*, which returns *true* iff r lies to the left of the oriented line through p and q.

PartitionIsValidTraits_2:: Orientation_2 — Predicate object type that provides *CGAL::Orientation operator()(Point_2 p, Point_2 q, Point_2 r)* that returns *CGAL::LEFT_TURN*, if r lies to the left of the oriented line l defined by p and q, returns *CGAL::RIGHT_TURN* if r lies to the right of l, and returns *CGAL::COLLINEAR* if r lies on l.

Creation

Only a copy constructor is required.

PartitionIsValidTraits_2 traits(& tr);

Operations

The following functions that create instances of the above predicate object types must exist.

Orientation_2	*traits.is_valid_object()*
Less_xy_2	*traits.less_xy_2_object()*
Left_turn_2	*traits.left_turn_2_object()*
Orientation_2	*traits.orientation_2_object()*

Has Models

CGAL::Partition_is_valid_traits_2<Traits, PolygonIsValid> . . . . page 1657

See Also

CGAL::approx_convex_partition_2 . . . . page 1630
CGAL::greene_approx_convex_partition_2 . . . . page 1636
CGAL::optimal_convex_partition_2 . . . . page 1646
CGAL::y_monotone_partition_2 . . . . page 1662

advanced

PartitionTraits_2

Definition

The polygon partitioning functions are each parameterized by a traits class that defines the primitives used in the algorithms. Many requirements are common to all traits classes. The concept PartitionTraits_2 defines this common set of requirements.

Types

PartitionTraits_2:: Point_2	The point type on which the partitioning algorithm operates.
PartitionTraits_2:: Polygon_2	The polygon type to be created by the partitioning algorithm. For testing the validity postcondition of the partition, this type should provide a nested type *Vertex_const_iterator* that is the type of the iterator over the polygon vertices and member functions *Vertex_const_iterator vertices_begin()* and *Vertex_const_iterator vertices_end()*.
PartitionTraits_2:: Less_xy_2	Predicate object type that compares *Point_2*s lexicographically. Must provide *bool operator()(Point_2 p, Point_2 q)* where *true* is returned iff $p <_{xy} q$. We have $p <_{xy} q$, iff $p_x < q_x$ or $p_x = q_x$ and $p_y < q_y$, where p_x and p_y denote the x and y coordinates of point p, respectively.
PartitionTraits_2:: Less_yx_2	Same as *Less_xy_2* with the roles of x and y interchanged.
PartitionTraits_2:: Left_turn_2	Predicate object type that provides *bool operator()(Point_2 p,Point_2 q,Point_2 r)*, which returns *true* iff r lies to the left of the oriented line through p and q.
PartitionTraits_2:: Orientation_2	Predicate object type that provides *CGAL::Orientation operator()(Point_2 p, Point_2 q, Point_2 r)* that returns *CGAL::LEFT_TURN*, if r lies to the left of the oriented line l defined by p and q, returns *CGAL::RIGHT_TURN* if r lies to the right of l, and returns *CGAL::COLLINEAR* if r lies on l.
PartitionTraits_2:: Compare_y_2	Predicate object type that provides *CGAL::Comparision_result operator()(Point_2 p, Point_2 q)* to compare the y values of two points. The operator must return *CGAL::SMALLER* if $p_y < q_y$, *CGAL::LARGER* if $p_y > q_y$ and *CGAL::EQUAL* if $p_y = q_y$.
PartitionTraits_2:: Compare_x_2	The same as *Compare_y_2*, except that x coordinates are compared instead of y.

Creation

A copy constructor and default constructor are required.

PartitionTraits_2 traits;

PartitionTraits_2 traits(& tr);

Operations

The following functions that create instances of the above predicate object types must exist.

Less_yx_2	*traits.less_yx_2_object()*
Less_xy_2	*traits.less_xy_2_object()*
Left_turn_2	*traits.left_turn_2_object()*
Orientation_2	*traits.orientation_2_object()*
Compare_y_2	*traits.compare_y_2_object()*
Compare_x_2	*traits.compare_x_2_object()*

Has Models

CGAL::Partition_traits_2<R> . . . . page 1659

See Also

CGAL::approx_convex_partition_2 . . . . page 1630
CGAL::greene_approx_convex_partition_2 . . . . page 1636
CGAL::optimal_convex_partition_2 . . . . page 1646
CGAL::y_monotone_partition_2 . . . . page 1662

advanced

CGAL::Partition_is_valid_traits_2<Traits, PolygonIsValid>

Class

Definition

Class that derives a traits class for *partition_is_valid_2* from a given traits class by defining the validity testing function object in terms of a supplied template parameter.

#include <CGAL/Partition_is_valid_traits_2.h>

Inherits From

Traits

Is Model for the Concepts

PartitionIsValidTraits_2

Types

typedef PolygonIsValid	*Is_valid;*
typedef Traits::Point_2	*Point_2;*
typedef Traits::Polygon_2	*Polygon_2;*
typedef Traits::Less_xy_2	*Less_xy_2;*
typedef Traits::Left_turn_2	*Left_turn_2;*
typedef Traits::Orientation_2	*Orientation_2;*

Operations

The constructors and member functions for creating instances of the above types are inherited from *Traits*. In addition, the following member function is defined:

Is_valid *traits.is_valid_object(Traits traits) const*

function returning an instance of *Is_valid*

See Also

CGAL::Is_convex_2<Traits> . . . . page 1643
CGAL::Is_vacuously_valid<Traits> . . . . page 1644
CGAL::Is_y_monotone_2<Traits> . . . . page 1645
CGAL::Partition_traits_2<R> . . . . page 1659

Example

See the example presented with the function *optimal_convex_partition_2* for an illustration of the use of this traits class.

advanced

CGAL::Partition_traits_2<R>

Class

Definition

Traits class that can be used with all the 2-dimensional polygon partitioning algorithms. It is parameterized by a representation class *R*.

#include <CGAL/Partition_traits_2.h>

Is Model for the Concepts

ConvexPartitionIsValidTraits_2 . . . . page 1635
IsYMonotoneTraits_2 . . . . page 1642
OptimalConvexPartitionTraits_2 . . . . page 1649
PartitionTraits_2 . . . . page 1655
YMonotonePartitionIsValidTraits_2 . . . . page 1667
YMonotonePartitionTraits_2 . . . . page 1668

Types

typedef R::Line_2	*Line_2;*
typedef R::Segment_2	*Segment_2;*
typedef R::Ray_2	*Ray_2;*
typedef R::Less_yx_2	*Less_yx_2;*
typedef R::Less_xy_2	*Less_xy_2;*
typedef R::Left_turn_2	*Left_turn_2;*
typedef R::Orientation_2	*Orientation_2;*
typedef R::Compare_y_2	*Compare_y_2;*
typedef R::Compare_x_2	*Compare_x_2;*
typedef R::Construct_line_2	*Construct_line_2;*
typedef R::Construct_ray_2	*Construct_ray_2;*
typedef R::Construct_segment_2	*Construct_segment_2;*
typedef R::Collinear_are_ordered_along_line_2	*Collinear_are_ordered_along_line_2;*
typedef R::Are_strictly_ordered_along_line_2	*Are_strictly_ordered_along_line_2;*
typedef CGAL::Polygon_traits_2<R>	*Poly_Traits;*
typedef Poly_Traits::Point_2	*Point_2;*
typedef std::list<Point_2>	*Container;*
typedef CGAL::Polygon_2<Poly_Traits, Container>	*Polygon_2;*
typedef R::Less_xy_2	*Less_xy;*
typedef Poly_Traits::Vector_2	*Vector_2;*
typedef R::FT	*FT;*
typedef Partition_traits_2<R>	*Self;*
typedef CGAL::Is_convex_2<Self>	*Is_convex_2;*
typedef CGAL::Is_y_monotone_2<Self>	*Is_y_monotone_2;*

Creation

A default constructor and copy constructor are defined.

Partition_traits_2<R> traits;

Partition_traits_2<R> traits(Partition_traits_2& tr);

Operations

For each predicate object type *Pred_object_type* listed above (*i.e.*, *Less_yx_2*, *Less_xy_2*, *Left_turn_2*, *Orientation_2*, *Compare_y_2*, *Compare_x_2*, *Construct_line_2*, *Construct_ray_2*, *Construct_segment_2*, *Collinear_are_ordered_along_line_2*, *Are_strictly_ordered_along_line_2*, *Is_convex_2*, *Is_y_monotone_2*) there is a corresponding function of the following form defined:

Pred_object_type	*traits.pred_object_type_object()*	Returns an instance of *Pred_object_type*.

See Also

CGAL::approx_convex_partition_2 . . . . page 1630
CGAL::convex_partition_is_valid_2 . . . . page 1633
CGAL::greene_approx_convex_partition_2 . . . . page 1636
CGAL::optimal_convex_partition_2 . . . . page 1646
CGAL::partition_is_valid_2 . . . . page 1651
CGAL::Partition_is_valid_traits_2<Traits, PolygonIsValid> . . . . page 1657
CGAL::y_monotone_partition_2 . . . . page 1662
CGAL::y_monotone_partition_is_valid_2 . . . . page 1665

advanced

PolygonIsValid

Functor
Concept

Definition

Function object that determines if a sequence of points represents a valid partition polygon or not, where “valid” can assume any of several meanings (*e.g.*, convex or *y*-monotone).

Creation

PolygonIsValid f(Traits t);

Traits is a model of the concept required by the function that checks for validity of the polygon.

Operations

template<class InputIterator>
bool f(InputIterator first, InputIterator beyond)

returns *true* iff the points of type *Traits::Point_2* in the range [*first*,*beyond*) define a valid polygon.

Has Models

CGAL::Is_convex_2<Traits> ... page 1643
CGAL::Is_y_monotone_2<Traits> ... page 1645

See Also

CGAL::approx_convex_partition_2 ... page 1630
CGAL::convex_partition_is_valid_2 ... page 1633
CGAL::greene_approx_convex_partition_2 ... page 1636
CGAL::optimal_convex_partition_2 ... page 1646
CGAL::partition_is_valid_2 ... page 1651
CGAL::y_monotone_partition_2 ... page 1662
CGAL::y_monotone_partition_is_valid_2 ... page 1665

advanced

Function

CGAL::y_monotone_partition_2

Definition

Function that produces a set of y-monotone polygons that represent a partitioning of a polygon defined on a sequence of points.

#include <CGAL/partition_2.h>

template <class InputIterator, class OutputIterator, class Traits>
OutputIterator y_monotone_partition_2(InputIterator first, InputIterator beyond, OutputIterator result, Traits traits = Default_traits)

computes a partition of the polygon defined by the points in the range [*first*, *beyond*) into y-monotone polygons. The counterclockwise-oriented partition polygons are written to the sequence starting at position *result*. The past-the-end iterator for the resulting sequence of polygons is returned.
Precondition: The points in the range [*first*, *beyond*) define a simple, counterclockwise-oriented polygon.

Requirements

1. *Traits* is a model of the concept *YMonotonePartitionTraits_2* and, for the purposes of checking the postcondition that the partition is valid, it should also be a model of *YMonotonePartitionIsValidTraits_2*.
2. *OutputIterator::value_type* should be *Traits::Polygon_2*.
3. *InputIterator::value_type* should be *Traits::Point_2*, which should also be the type of the points stored in an object of type *Traits::Polygon_2*.

The default traits class *Default_traits* is *Partition_traits_2*, with the representation type determined by *InputIterator::value_type*.

See Also

CGAL::approx_convex_partition_2 . . . . . page 1630
CGAL::greene_approx_convex_partition_2 . . . . . page 1636
CGAL::optimal_convex_partition_2 . . . . . page 1646
CGAL::partition_is_valid_2 . . . . . page 1651
CGAL::y_monotone_partition_is_valid_2 . . . . . page 1665

Implementation

This function implements the algorithm presented by de Berg *et al.* [dBvKOS97] which requires $O(n \log n)$ time and $O(n)$ space for a polygon with n vertices.

Example

The following program computes a y-monotone partitioning of a polygon using the default traits class and stores the partition polygons in the list *partition_polys*. It then asserts that each partition polygon produced is, in fact, y-monotone and that the partition is valid. (Note that these assertions are superfluous unless the postcondition checking for *y_monotone_partition_2* has been turned off.)

```
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/Partition_traits_2.h>
#include <CGAL/partition_2.h>
#include <CGAL/point_generators_2.h>
#include <CGAL/random_polygon_2.h>
#include <cassert>
#include <list>


typedef CGAL::Exact_predicates_inexact_constructions_kernel K;
typedef CGAL::Partition_traits_2<K>                         Traits;
typedef Traits::Point_2                                     Point_2;
typedef Traits::Polygon_2                                   Polygon_2;
typedef std::list<Polygon_2>                                Polygon_list;
typedef CGAL::Creator_uniform_2<int, Point_2>               Creator;
typedef CGAL::Random_points_in_square_2<Point_2, Creator>
   Point_generator;

void make_polygon(Polygon_2& polygon)
{
   polygon.push_back(Point_2(391, 374));
   polygon.push_back(Point_2(240, 431));
   polygon.push_back(Point_2(252, 340));
   polygon.push_back(Point_2(374, 320));
   polygon.push_back(Point_2(289, 214));
   polygon.push_back(Point_2(134, 390));
   polygon.push_back(Point_2( 68, 186));
   polygon.push_back(Point_2(154, 259));
   polygon.push_back(Point_2(161, 107));
   polygon.push_back(Point_2(435, 108));
   polygon.push_back(Point_2(208, 148));
   polygon.push_back(Point_2(295, 160));
   polygon.push_back(Point_2(421, 212));
   polygon.push_back(Point_2(441, 303));
}


int main( )
{
   Polygon_2    polygon;
   Polygon_list partition_polys;

/*
   CGAL::random_polygon_2(50, std::back_inserter(polygon),
                          Point_generator(100));
*/
   make_polygon(polygon);
   CGAL::y_monotone_partition_2(polygon.vertices_begin(),
```

```
                                   polygon.vertices_end(),
                                   std::back_inserter(partition_polys));

   std::list<Polygon_2>::const_iterator   poly_it;
   for (poly_it = partition_polys.begin(); poly_it !=
   partition_polys.end();
        poly_it++)
   {
      assert(CGAL::is_y_monotone_2((*poly_it).vertices_begin(),
                                   (*poly_it).vertices_end()));
   }

   assert(CGAL::partition_is_valid_2(polygon.vertices_begin(),
                                     polygon.vertices_end(),
                                     partition_polys.begin(),
                                     partition_polys.end()));
   return 0;
}
```

File: examples/Partition_2/y_monotone_partition_2.cpp

advanced

CGAL::y_monotone_partition_is_valid_2

Function

Definition

Function that determines if a given set of polygons represents a valid y-monotone partitioning for a given sequence of points that define a simple, counterclockwise-oriented polygon. A valid partition is one in which the polygons are nonoverlapping and the union of the polygons is the same as the original polygon and each polygon is y-monotone

#include <CGAL/partition_is_valid_2.h>

```
template<class InputIterator, class ForwardIterator, class Traits>
bool    y_monotone_partition_is_valid_2( InputIterator point_first,
                                         InputIterator point_beyond,
                                         ForwardIterator poly_first,
                                         ForwardIterator poly_beyond,
                                         Traits traits = Default_traits)
```

determines if the polygons in the range [*poly_first*, *poly_beyond*) define a valid y-monotone partition of the polygon represented by the points in the range [*point_first*, *point_beyond*). The function returns *true* iff the partition is valid and otherwise returns false.
Precondition: Points in the range [*point_first*, *point_beyond*) define a simple, counterclockwise-oriented polygon.

Requirements

1. *Traits* is a model of the concept *YMonotonePartitionIsValidTraits_2*.
2. *InputIterator::value_type* should be *Traits::Point_2*, which should also be the type of the points stored in an object of type *Traits::Polygon_2*.
3. *ForwardIterator::value_type* should be *Traits::Polygon_2*.

The default traits class *Default_traits* is *Partition_traits_2*, with the representation type determined by *InputIterator::value_type*.

See Also

CGAL::y_monotone_partition_2 . . . . . page 1662
CGAL::is_y_monotone_2 . . . . . page 1639
CGAL::partition_is_valid_2 . . . . . page 1651
CGAL::Partition_is_valid_traits_2<Traits, PolygonIsValid> . . . . . page 1657

Implementation

This function uses the function *partition_is_valid_2* together with the function object *Is_y_monotone_2* to determine if each polygon is y-monotone or not. Thus the time required is $O(n \log n + e \log e)$ where n is the total number of vertices of the partition polygons and e is the total number of edges.

Example

See the example presented with the function *y_monotone_partition_2* for an illustration of the use of this function.

advanced

advanced

YMonotonePartitionIsValidTraits_2

Concept

Definition

Requirements of a traits class that is used by *y_monotone_partition_is_valid_2* for testing the validity of a *y*-monotone partition of a polygon.

Types

All types required by the concept *PartitionIsValidTraits_2* are required except the function object type *Is_valid*. The following type is required instead:

YMonotonePartitionIsValidTraits_2:: Is_y_monotone_2

Model of the concept *PolygonIsValid* that tests if a sequence of points is *y*-monotone or not.

Creation

Only a copy constructor is required.

YMonotonePartitionIsValidTraits_2 traits(& tr);

Operations

The following function that creates an instance of the above predicate object type must exist instead of the function *is_valid_object* required by *PartitionIsValidTraits_2*.

Is_y_monotone_2   *traits.is_y_monotone_2_object(t)*

Has Models

CGAL::Partition_traits_2<R> . . . . page 1659

See Also

CGAL::partition_is_valid_2 . . . . page 1651
CGAL::y_monotone_partition_2 . . . . page 1662

advanced

Concept

YMonotonePartitionTraits_2

Definition

Requirements of a traits class to be used with the function *y_monotone_partition_2*.

Refines

PartitionTraits_2 . . . . page 1655

Types

In addition to the types defined for the concept *PartitionTraits_2*, the following types are also required:

YMonotonePartitionTraits_2:: Line_2

YMonotonePartitionTraits_2:: Compare_x_at_y_2

Predicate object type that provides *CGAL::Comparision_result operator()(Point_2 p, Line_2 h)* to compare the x coordinate of p and the horizontal projection of p on h.

YMonotonePartitionTraits_2:: Construct_line_2

Function object type that provides *Line_2 operator()(Point_2 p, Point_2 q)*, which constructs and returns the line defined by the points p and q.

YMonotonePartitionTraits_2:: Is_horizontal_2 Function object type that provides *bool operator()(Line_2 l)*, which returns *true* iff the line l is horizontal.

Creation

A copy constructor and default constructor are required.

YMonotonePartitionTraits_2 traits;

YMonotonePartitionTraits_2 traits(YMonotonePartitionTraits tr);

Operations

In addition to the functions required for the concept *PartitionTraits_2*, the following functions that create instances of the above function object types must exist.

Construct_line_2 *traits.construct_line_2_object()*

Compare_x_at_y_2 *traits.compare_x_at_y_2_object()*

Is_horizontal_2 *traits.is_horizontal_2_object()*

Has Models

CGAL::Partition_traits_2<R> . . . . page 1659

Chapter 23

2D Straight Skeleton and Polygon Offsetting

Fernando Cacciola

Contents

23.1 Definitions

23.1.1 2D Contour

A *2D contour* is a closed sequence (a cycle) of 3 or more *connected 2D oriented straight line segments* called *contour edges*. The endpoints of the contour edges are called *vertices*. Each contour edge shares its endpoints with at least two other contour edges.

If the edges intersect only at the vertices and at most are coincident along a line but do not *cross* one another, the contour is classified as *simple*.
A contour is topologically equivalent to a *disk* and if it is simple, is said to be a *Jordan Curve*.
Contours partition the plane in two open regions: one bounded and one unbounded. If the bounded region of a contour is only one *singly-connected set*, the contour is said to be *strictly-simple*.
The *Orientation* of a contour is given by the order of the vertices around the region they bound. It can be *CLOCKWISE* (CCW) or *COUNTERCLOCKWISE* (CW).
The *bounded side* of a contour edge is the side facing the bounded region of the contour. If the contour is oriented CCW, the bounded side of an edge is its left side.

A contour with a null edge (a segment of length zero given by two consecutive coincident vertices), or with edges not connected to the bounded region (an antenna: 2 consecutive edges going forth and back along the same line), is said to be *degenerate* (collinear edges are *not* considered a degeneracy).

23.1.2 2D Polygon with Holes

A *2D polygon* is a contour.
A *2D polygon with holes* is a contour, called the *outer contour*, having zero or more contours, called *inner contours*, or *holes*, in its bounded region. The intersection of the bounded region of the outer contour and the unbounded regions of each inner contour is the *interior* of the polygon with holes. The orientation of the holes must be opposite to the orientation of the outer contour and there cannot be any intersection among any contour. A hole cannot be in the bounded region of any other hole.
A polygon with holes is strictly-simple if its interior is a singly-connected set.
The orientation of a polygon with holes is the orientation of its outer contour. The bounded side of *any* edge, whether of the outer contour or a hole, is the *same* for all edges. That is, if the outer contour is oriented CCW and the holes CW, both contour and hole edges face the polygon interior to their left.

Throughout the rest of this chapter the term *polygon* will be used as a shortcut for *polygon with holes*.

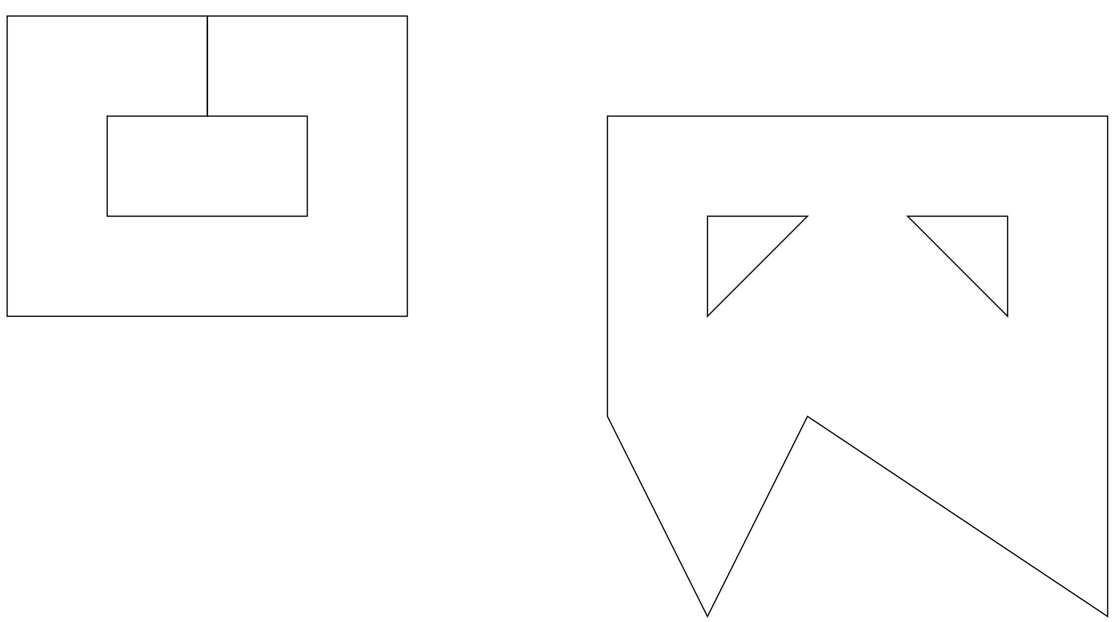


Figure 23.1: Examples of strictly simple polygons: One with no holes and two edges coincident (left) and one with 2 holes (right).

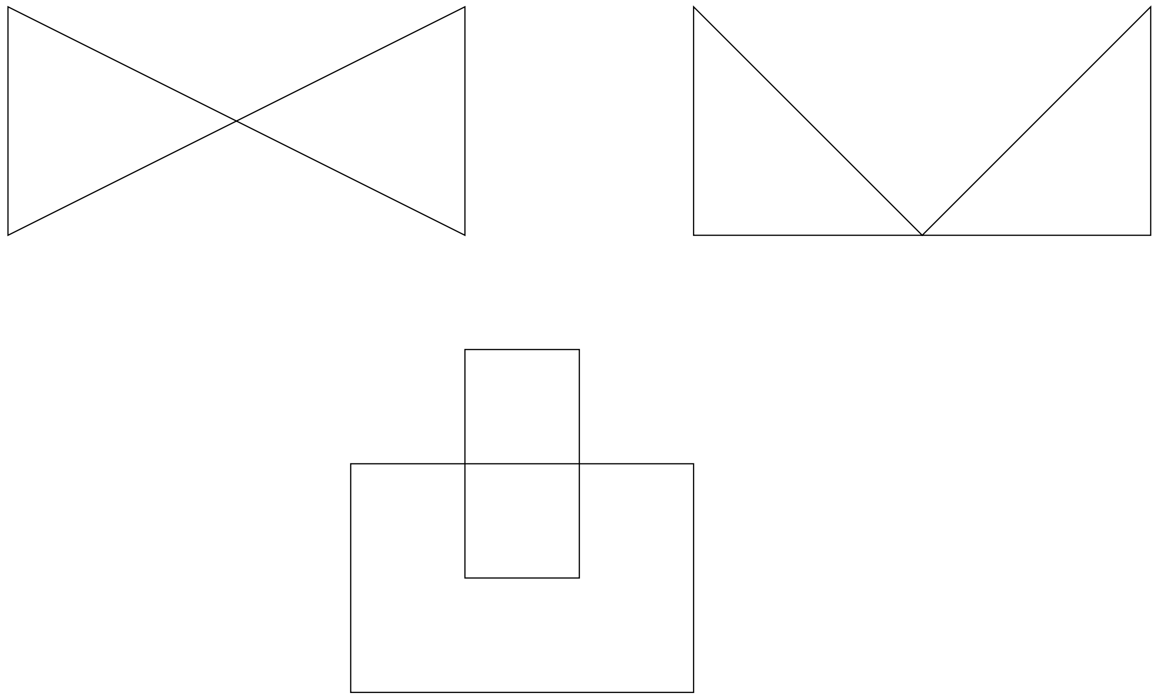


Figure 23.2: Examples of non-simple polygons: One folding into itself, that is, non-planar (left), one with a vertex touching an edge (right), and one with a hole crossing into the outside (bottom)

23.1.3 Inward Offset of a Non-degenerate Strictly-Simple Polygon with Holes

For any 2D non-degenerate strictly-simple polygon with holes called the *source*, there can exist a *set* of 0, 1 or more *inward offset polygons with holes*, or just offset polygons for short, at some euclidean distance $t > 0$ (each being strictly simple and non-degenerate). Any contour edge of such offset polygon, called an *offset edge* corresponds to *some* contour edge of the source polygon, called its *source edge*. An offset edge is parallel to its source edge and has the same orientation. The Euclidean distance between the *lines* supporting an offset edge and its source edge is exactly t.

An offset edge is always located to the bounded side of its source edge (which is an oriented straight line segment).

An offset polygon can have less, equal or more sides as its source polygon.

If the source polygon has no holes, no offset polygon has holes. If the source polygon has holes, any of the offset polygons can have holes itself, but it might as well have no holes at all (if the distance is sufficiently large).

Each offset polygon has the same orientation as the source polygon.

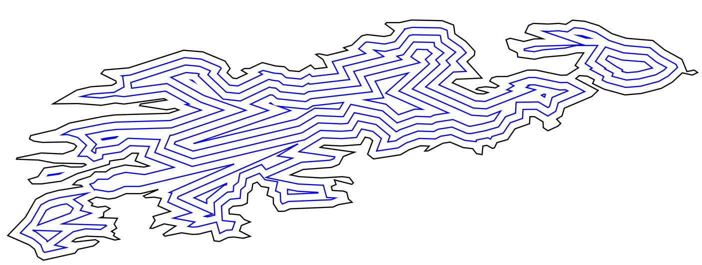


Figure 23.3: Offset contours of a sample polygon

23.1.4 Straight Skeleton of a 2D Non-degenerate Strictly-Simple Polygon with Holes

The *2D straight skeleton* of a non-degenerate strictly-simple polygon with holes [AAAG95] is a special partitioning of the polygon interior into *straight skeleton regions* corresponding to the monotone areas traced by a continuous *inward offsetting* of the contour edges. Each region corresponds to exactly 1 contour edge.
These regions are bounded by angular bisectors of the supporting lines of the contour edges and each such region is itself a non-convex non-degenerate strictly-simple polygon.

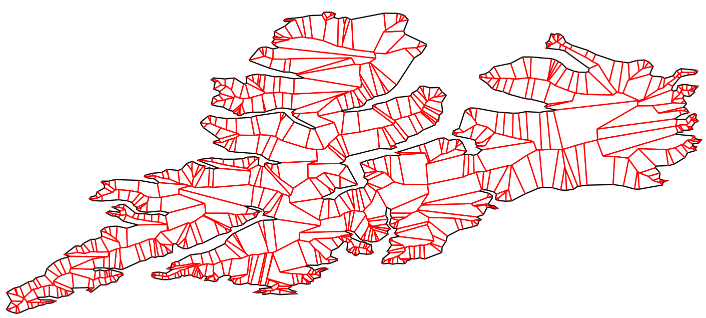


Figure 23.4: Straight skeleton of a complex shaggy contour

Angular Bisecting Lines and Offset Bisectors

Given two points and a line passing through them, the perpendicular line passing through the midpoint is the bisecting line (or bisector) of those points.

Two non-parallel lines, intersecting at a point, are bisected by two other lines passing through that intersection point.

Two parallel lines are bisected by another parallel line placed halfway in between.

Given just one line, any perpendicular line can be considered the bisecting line (any bisector of any two points along the single line).

The bisecting lines of two edges are the lines bisecting the supporting lines of the edges (if the edges are parallel or collinear, there is just one bisecting line).

The halfplane to the bounded side of the line supporting a contour edge is called the *offset zone* of the contour edge.

Given any number of contour edges (not necessarily consecutive), the intersection of their offset zones is called their *combined offset zone*.

Any two contour edges define an *offset bisector*, as follows: If the edges are non-parallel, their bisecting lines can be decomposed as 4 rays originating at the intersection of the supporting lines. Only one of these rays is contained in the combined offset zone of the edges (which one depends on the possible combinations of orientations). This ray is the offset bisector of the non-parallel contour edges.

If the edges are parallel (but not collinear) and have opposite orientation, the entire and unique bisecting line is their offset bisector. If the edges are parallel but have the same orientation, there is no offset bisector between them.

If the edges are collinear and have the same orientation, their offset bisector is given by a perpendicular ray to the left of the edges which originates at the midpoint of the combined complement of the edges. (The *complement*

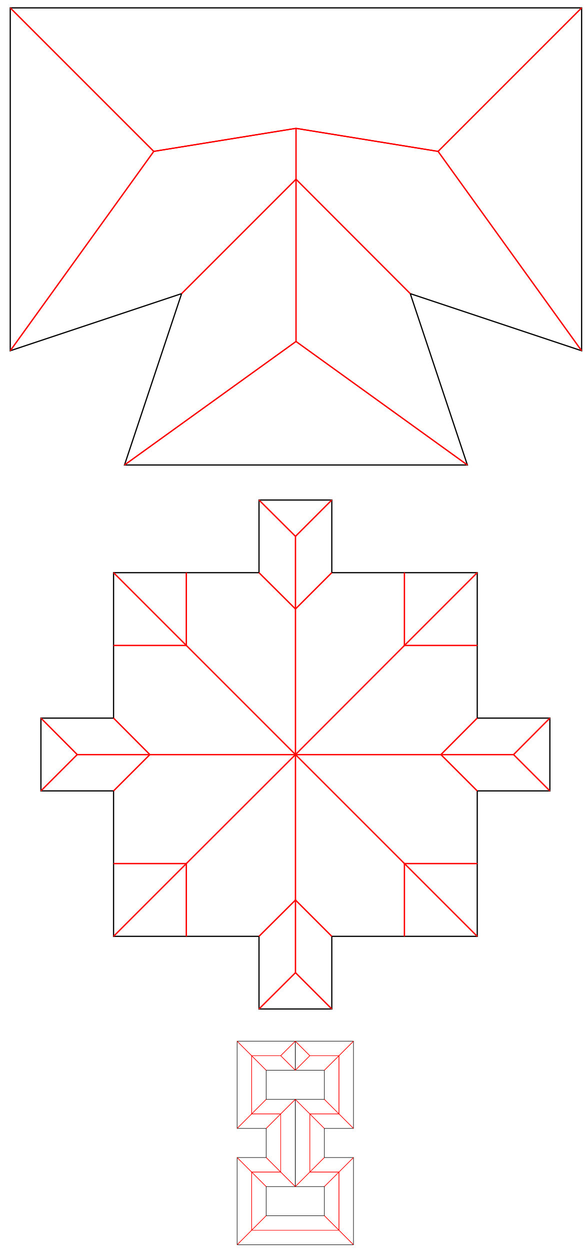


Figure 23.5: Other examples: A vertex-event (left), the case of several collinear edges (middle), and the case of a validly simple polygon with tangent edges (right).

of an edge/segment are the two rays along its supporting line which are not the segment and the *combined complement* of N collinear segments is the intersection of the complements of each segment). If the edges are collinear but have opposite orientation, there is no offset bisector between them.

Faces, Edges and Vertices

Each region of the partitioning defined by a straight skeleton is called a *face*. Each face is bounded by straight line segments, called *edges*. Exactly one edge per face is a *contour edge* (corresponds to a side of the polygon) and the rest of the edges, located in the interior of the polygon, are called *skeleton edges*, or *bisectors*.

The bisectors of the straight skeleton are segments of the offset bisectors as defined previously. Since an offset bisector is a ray of a bisecting line of 2 contour edges, each skeleton edge (or bisector) is uniquely given by two contour edges. These edges are called the *defining contour edges* of the bisector.

The intersection of the edges are called *vertices*. Although in a simple polygon, only 2 edges intersect at a vertex, in a straight skeleton, 3 or more edges intersect a any given vertex. That is, vertices in a straight skeleton have degree $>= 3$.

A *contour vertex* is a vertex for which 2 of its incident edges are contour edges.

A *skeleton vertex* is a vertex who's incident edges are all skeleton edges.

A *contour bisector* is a bisector who's defining contour edges are consecutive. Such a bisector is incident upon 1 contour vertex and 1 skeleton vertex and touches the input polygon at exactly 1 endpoint.

An *inner bisector* is a bisector who's defining contour edges are not consecutive. Such a bisector is incident upon 2 skeleton vertices and is strictly contained in the interior of the polygon.

23.2 Representation

This CGAL package represents a straight skeleton as a specialized *Halfedge Data Structure* (HDS) whose vertices embeds 2D Points (see the *StraightSkeleton_2* concept in the reference manual for details).

Its halfedges, by considering the source and target points, implicitly embeds 2D oriented straight line segments (each halfedge per see does not embed a segment explicitly).

A face of the straight skeleton is represented as a face in the HDS. Both contour and skeleton edges are represented by pairs of opposite HDS halfedges, and both contour and skeleton vertices are represented by HDS vertices.

In a HDS, a border halfedge is a halfedge which is incident upon an unbounded face. In the case of the straight skeleton HDS, such border halfedges are oriented such that their left side faces outwards the polygon. Therefore, the opposite halfedge of any border halfedge is oriented such that its left side faces inward the polygon.

This CGAL package requires the input polygon (with holes) to be non-degenerate, strictly-simple, and oriented counter-clockwise.

The skeleton halfedges are oriented such that their *left* side faces inward the region they bound. That is, the vertices (both contour and skeleton) of a face are circulated in counter-clockwise order. There is one and only one contour halfedge incident upon any face.

The contours of the input polygon are traced by the border halfedges of the HDS (those facing outward), but in the opposite direction. That is, the vertices of the contours can only by traced from the straight skeleton data structure by circulating the border halfedges, and the resulting vertex sequence will be reversed w.r.t the input vertex sequence.

A skeleton edge, according to the definition given in the previous section, is defined by 2 contour edges. In the representation, each one of the opposite halfedges that represent a skeleton edge is associated with one of the opposite halfedges that correspond to one of its defining contour edges. Thus, the 2 opposite halfedges of a skeleton edge link the edge to its 2 defining contour edges.

Starting from any border contour halfedge, circulating the structure walks through border counter halfedges and traces the vertices of the polygon's contours (in opposite order).

Starting from any non-border but contour halfedge, circulating the structure walks counter-clockwise around the face corresponding to that contour halfedge. The vertices around a face always describe a non-convex non-degenerate strictly-simple polygon.

A vertex is the intersection of contour and/or skeleton edges. Since a skeleton edge is defined by 2 contour edges, any vertex is itself defined by a unique set of contour edges. These are called the *defining contour edges* of the vertex.

A vertex is identified by it's set of defining contour edges. Two vertices are distinct if they have differing sets of defining contour edges. Note that vertices can be distinct even if they are geometrically embedded at the same point.

The *degree* of a vertex is the number of halfedges around the vertex incident upon (pointing to) the vertex. As with any halfedge data structure, there is one outgoing halfedge for each incoming (incident) halfedge around a vertex. The degree of the vertex counts only incoming (incident) halfedges.

In a straight skeleton, the degree of a vertex is not only the number of incident halfedges around the vertex but also the number of defining contour halfedges. The vertex itself is the point where all the defining contour edges simultaneously collide.

Contour vertices have exactly two defining contour halfedges, which are the contour edges incident upon the vertex; and 3 incident halfedges. One and only one of the incident halfedges is a skeleton halfedge. The degree of a contour vertex is exactly 3.

Skeleton vertices have at least 3 defining contour halfedges and 3 incident skeleton halfedges. If more than 3 edges collide simultaneously at the same point and time (like in any regular polygon with more than 3 sides), the corresponding skeleton vertex will have more than 3 defining contour halfedges and incident skeleton halfedges. That is, the degree of a skeleton vertex is $>= 3$ (the algorithm initially produces nodes of degree 3 but in the end all coincident nodes are merged to form higher degree nodes). All halfedges incident upon a skeleton vertex are skeleton halfedges.

The defining contour halfedges and incident halfedges around a vertex can be traced using the circulators provided by the vertex class. The degree of a vertex is not cached and cannot be directly obtained from the vertex, but you can calculate this number by manually counting the number of incident halfedges around the vertex.

Each vertex stores a 2D point and a *time*, which is the euclidean distance from the vertex's point to the *lines* supporting each of the defining contour edges of the vertex (the distance is the same to each line). Unless the polygon is convex, this distance is not equidistant to the edges, as in the case of a Medial Axis, therefore, the time of a skeleton vertex does not correspond to the distance from the polygon to the vertex (so it cannot be used to obtain the deep of a region in a shape, for instance).
If the polygon *is* convex, the straight skeleton is exactly equivalent to the polygon's Voronoi diagram and each vertex time is the equidistance to the defining edges.
Contour vertices have time zero.

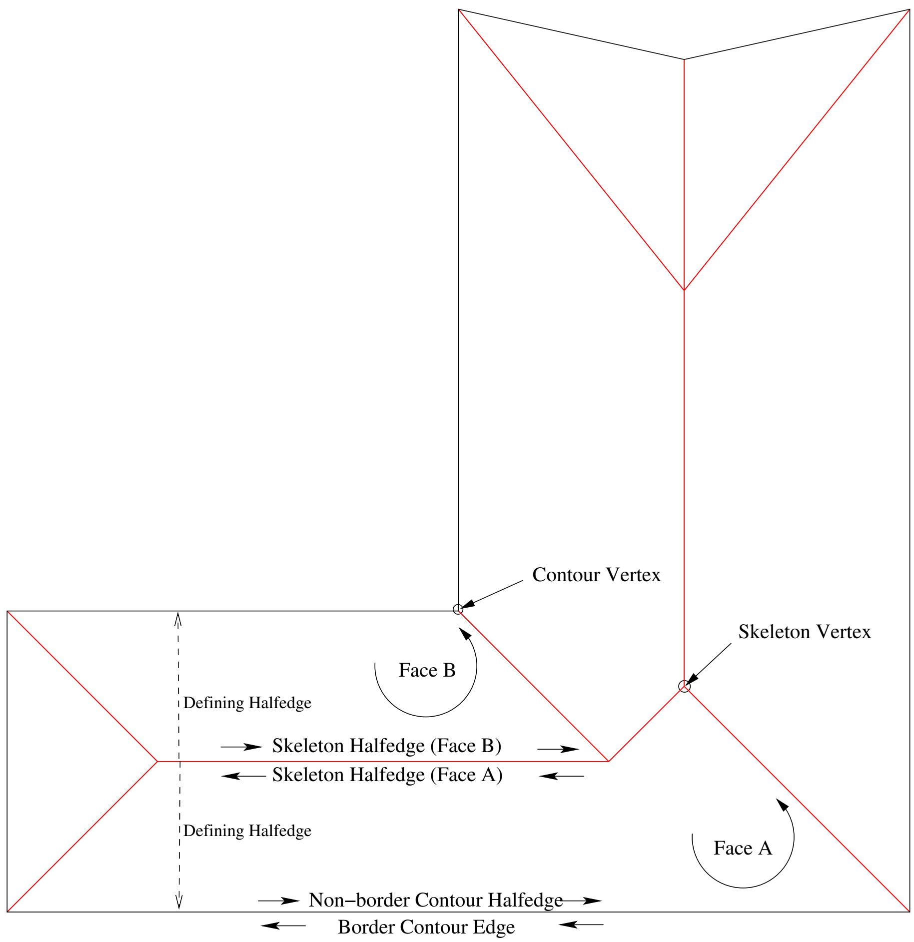


Figure 23.6: Straight Skeleton Data Structure

23.3 API

23.3.1 Create a Straight Skeleton

The straight skeleton data structure is defined by the *StraightSkeleton_2* concept and modeled in the *Straight_skeleton_2<Traits,Items,Alloc>* class.

The simplest way to construct a straight skeleton is via the free functions *create_interior_straight_skeleton_2* and *create_exterior_straight_skeleton_2*, as shown in the following example:

```
#include<boost/shared_ptr.hpp>

#include<CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include<CGAL/Polygon_2.h>
#include<CGAL/create_straight_skeleton_2.h>

#include "print.h"

typedef CGAL::Exact_predicates_inexact_constructions_kernel K ;

typedef K::Point_2                    Point ;
typedef CGAL::Polygon_2<K>            Polygon_2 ;
typedef CGAL::Straight_skeleton_2<K> Ss ;

typedef boost::shared_ptr<Ss> SsPtr ;

int main()
{
  Polygon_2 poly ;

  poly.push_back( Point(-1,-1) ) ;
  poly.push_back( Point(0,-12) ) ;
  poly.push_back( Point(1,-1) ) ;
  poly.push_back( Point(12,0) ) ;
  poly.push_back( Point(1,1) ) ;
  poly.push_back( Point(0,12) ) ;
  poly.push_back( Point(-1,1) ) ;
  poly.push_back( Point(-12,0) ) ;

  // You can pass the polygon via an iterator pair
  SsPtr iss =
   CGAL::create_interior_straight_skeleton_2(poly.vertices_begin(),
   poly.vertices_end());

  // Or you can pass the polygon directly, as below.

  // To create an exterior straight skeleton you need to specify a
   maximum offset.
  double lMaxOffset = 5 ;
  SsPtr oss = CGAL::create_exterior_straight_skeleton_2(lMaxOffset,
   poly);

  print_straight_skeleton(*iss);
  print_straight_skeleton(*oss);
```

```
  return 0;
}
```

File: examples/Straight_skeleton_2/Create_straight_skeleton_2.cpp

The input to these functions is the polygon, which can be given as an iterator pair or directly as a *Polygon_2* object. In the case of the exterior skeleton, a maximum offset must be specified as well (see the ref manual for details on this max offset parameter).

23.3.2 Create a Straight Skeleton from a Polygon With Holes

If *Polygon_with_holes_2* is used, you can pass an instance of it directly to the function creating the interior skeleton, as shown below. Notice that a different header must be included in this case.

```
#include<boost/shared_ptr.hpp>

#include<CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include<CGAL/Polygon_with_holes_2.h>
#include<CGAL/create_straight_skeleton_from_polygon_with_holes_2.h>

#include "print.h"

typedef CGAL::Exact_predicates_inexact_constructions_kernel K ;

typedef K::Point_2                    Point ;
typedef CGAL::Polygon_2<K>            Polygon_2 ;
typedef CGAL::Polygon_with_holes_2<K> Polygon_with_holes ;
typedef CGAL::Straight_skeleton_2<K>  Ss ;

typedef boost::shared_ptr<Ss> SsPtr ;


int main()
{
  Polygon_2 outer ;

  outer.push_back( Point(-1,-1) ) ;
  outer.push_back( Point(0,-12) ) ;
  outer.push_back( Point(1,-1) ) ;
  outer.push_back( Point(12,0) ) ;
  outer.push_back( Point(1,1) ) ;
  outer.push_back( Point(0,12) ) ;
  outer.push_back( Point(-1,1) ) ;
  outer.push_back( Point(-12,0) ) ;

  Polygon_2 hole ;

  hole.push_back( Point(-1,0) ) ;
  hole.push_back( Point(0,1 ) ) ;
  hole.push_back( Point(1,0 ) ) ;
```

```
  hole.push_back( Point(0,-1) ) ;

  Polygon_with_holes poly( outer ) ;

  poly.add_hole( hole ) ;

  SsPtr iss = CGAL::create_interior_straight_skeleton_2(poly);

  print_straight_skeleton(*iss);

  return 0;
}
```

File: examples/Straight_skeleton_2/Create_straight_skeleton_from_polygon_with_holes_2.cpp

23.3.3 Create Offset Polygons from a Straight Skeleton

If you already have a straight skeleton instance, the simpler way to generate offset polygons is to call *create_offset_polygons_2* as shown in the next example, passing the desired offset and the straight skeleton. You can reuse the same skeleton to generate offsets at a different distance, which is recommended because producing the straight skeleton is much slower then generating offset polygons.

```
#include<vector>

#include<boost/shared_ptr.hpp>

#include<CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include<CGAL/Polygon_2.h>
#include<CGAL/create_offset_polygons_2.h>

#include "print.h"

typedef CGAL::Exact_predicates_inexact_constructions_kernel K ;

typedef K::Point_2                   Point ;
typedef CGAL::Polygon_2<K>           Polygon_2 ;
typedef CGAL::Straight_skeleton_2<K> Ss ;

typedef boost::shared_ptr<Polygon_2> PolygonPtr ;
typedef boost::shared_ptr<Ss> SsPtr ;

typedef std::vector<PolygonPtr> PolygonPtrVector ;

int main()
{
  Polygon_2 poly ;

  poly.push_back( Point(-1,-1) ) ;
  poly.push_back( Point(0,-12) ) ;
  poly.push_back( Point(1,-1) ) ;
  poly.push_back( Point(12,0) ) ;
  poly.push_back( Point(1,1) ) ;
```

```
  poly.push_back( Point(0,12) ) ;
  poly.push_back( Point(-1,1) ) ;
  poly.push_back( Point(-12,0) ) ;

  SsPtr ss = CGAL::create_interior_straight_skeleton_2(poly);

  double lOffset = 1 ;

  PolygonPtrVector offset_polygons =
   CGAL::create_offset_polygons_2<Polygon_2>(lOffset,*ss);

  print_polygons(offset_polygons);

  return 0;
}
```

File: examples/Straight_skeleton_2/Create_offset_polygons_2.cpp

23.3.4 Create Offset Polygons from a Polygon (With or Without Holes)

If you need offset polygons at just one single distance, you can hide away the construction of the straight skeleton by calling directly the functions *create_interior_skeleton_and_offset_polygons_2* and *create_exterior_skeleton_and_offset_polygons_2* as shown in the following examples:

```
#include<vector>

#include<boost/shared_ptr.hpp>

#include<CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include<CGAL/Polygon_2.h>
#include<CGAL/create_offset_polygons_2.h>

#include "print.h"

typedef CGAL::Exact_predicates_inexact_constructions_kernel K ;

typedef K::FT                        FT ;
typedef K::Point_2                   Point ;
typedef CGAL::Polygon_2<K>           Polygon_2 ;
typedef CGAL::Straight_skeleton_2<K> Ss ;

typedef boost::shared_ptr<Polygon_2> PolygonPtr ;
typedef boost::shared_ptr<Ss> SsPtr ;

typedef std::vector<PolygonPtr> PolygonPtrVector ;


int main()
{
  Polygon_2 poly ;

  poly.push_back( Point(-1,-1) ) ;
```

```
  poly.push_back( Point(0,-12) ) ;
  poly.push_back( Point(1,-1) ) ;
  poly.push_back( Point(12,0) ) ;
  poly.push_back( Point(1,1) ) ;
  poly.push_back( Point(0,12) ) ;
  poly.push_back( Point(-1,1) ) ;
  poly.push_back( Point(-12,0) ) ;

  FT lOffset = 1 ;

  PolygonPtrVector inner_offset_polygons =
   CGAL::create_interior_skeleton_and_offset_polygons_2(lOffset,poly);
  PolygonPtrVector outer_offset_polygons =
   CGAL::create_exterior_skeleton_and_offset_polygons_2(lOffset,poly);

  print_polygons(inner_offset_polygons);
  print_polygons(outer_offset_polygons);

  return 0;
}
```

File: examples/Straight_skeleton_2/Create_skeleton_and_offset_polygons_2.cpp

... and using a *Polygon_with_holes_2* directly when available:

```
#include<vector>
#include<iterator>
#include<iostream>
#include<iomanip>
#include<string>

#include<boost/shared_ptr.hpp>

#include<CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include<CGAL/Polygon_with_holes_2.h>
#include<CGAL/create_offset_polygons_from_polygon_with_holes_2.h>

#include "print.h"

typedef CGAL::Exact_predicates_inexact_constructions_kernel K ;

typedef K::Point_2                    Point ;
typedef CGAL::Polygon_2<K>            Polygon_2 ;
typedef CGAL::Polygon_with_holes_2<K> Polygon_with_holes ;
typedef CGAL::Straight_skeleton_2<K>  Ss ;

typedef boost::shared_ptr<Polygon_2> PolygonPtr ;
typedef boost::shared_ptr<Ss> SsPtr ;

typedef std::vector<PolygonPtr> PolygonPtrVector ;


int main()
```

```
{
  Polygon_2 outer ;

  outer.push_back( Point(-1,-1) ) ;
  outer.push_back( Point(0,-12) ) ;
  outer.push_back( Point(1,-1) ) ;
  outer.push_back( Point(12,0) ) ;
  outer.push_back( Point(1,1) ) ;
  outer.push_back( Point(0,12) ) ;
  outer.push_back( Point(-1,1) ) ;
  outer.push_back( Point(-12,0) ) ;

  Polygon_2 hole ;

  hole.push_back( Point(-1,0) ) ;
  hole.push_back( Point(0,1 ) ) ;
  hole.push_back( Point(1,0 ) ) ;
  hole.push_back( Point(0,-1) ) ;

  Polygon_with_holes poly( outer ) ;

  poly.add_hole( hole ) ;

  double lOffset = 0.2 ;

  PolygonPtrVector offset_polygons =
   CGAL::create_interior_skeleton_and_offset_polygons_2(lOffset,poly);

  print_polygons(offset_polygons);

  return 0;
}
```

File: examples/Straight_skeleton_2/Create_saop_from_polygon_with_holes_2.cpp

If the input polygon has holes, there can be holes in the offset polygons. However, the polygons generated by all the offsetting functions shown before do not have any parent-hole relationship computed; that is, they just instances of *Polygon_2* instead of *Polygon_with_holes_2*. If *Polygon_with_holes_2* are available and you need the offsetting to produce them, you can call the function *arrange_offset_polygons_2* passing the result of any of the offsetting functions described so far. That function arranges the offset polygons detecting and distributing holes within parents. As a shortcut, you can use the function *create_interior_skeleton_and_offset_polygon_with_holes_2* as shown below:

```
#include<vector>

#include<boost/shared_ptr.hpp>

#include<CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include<CGAL/Polygon_with_holes_2.h>
#include<CGAL/create_offset_polygons_from_polygon_with_holes_2.h>

#include "print.h"
```

```
typedef CGAL::Exact_predicates_inexact_constructions_kernel K ;

typedef K::Point_2                    Point ;
typedef CGAL::Polygon_2<K>            Polygon_2 ;
typedef CGAL::Polygon_with_holes_2<K> PolygonWithHoles ;
typedef CGAL::Straight_skeleton_2<K>  Ss ;

typedef boost::shared_ptr<PolygonWithHoles> PolygonWithHolesPtr ;

typedef std::vector<PolygonWithHolesPtr> PolygonWithHolesPtrVector;


int main()
{
  Polygon_2 outer ;

  outer.push_back( Point( 0.0, 0.0) ) ;
  outer.push_back( Point(10.0, 0.0) ) ;
  outer.push_back( Point(10.0, 4.5) ) ;
  outer.push_back( Point(12.0, 4.5) ) ;
  outer.push_back( Point(12.0, 2.0) ) ;
  outer.push_back( Point(16.0, 2.0) ) ;
  outer.push_back( Point(16.0, 8.0) ) ;
  outer.push_back( Point(12.0, 8.0) ) ;
  outer.push_back( Point(12.0, 5.5) ) ;
  outer.push_back( Point(10.0, 5.5) ) ;
  outer.push_back( Point(10.0,10.0) ) ;
  outer.push_back( Point( 0.0,10.0) ) ;

  Polygon_2 hole ;

  hole.push_back( Point(3.0,3.0) ) ;
  hole.push_back( Point(3.0,7.0) ) ;
  hole.push_back( Point(7.0,7.0) ) ;
  hole.push_back( Point(7.0,3.0) ) ;

  PolygonWithHoles poly( outer ) ;

  poly.add_hole( hole ) ;

  double lOffset = 1 ;

  PolygonWithHolesPtrVector offset_poly_with_holes =
   CGAL::create_interior_skeleton_and_offset_polygons_with_holes_2(lOffset,poly);

  print_polygons_with_holes(offset_poly_with_holes);

  return 0;
}
```

File: examples/Straight_skeleton_2/Create_skeleton_and_offset_polygons_with_holes_2.cpp

advanced

Consider an input polygon with parallel edges separated a distance $2*t$. If you produce an offset polygon at distance t, these parallel edges will just collapse each other and vanish from the result, keeping the output as a *simple polygon*, just like the input. However, if you request an offset polygon at a distance $t-epsilon$, the result will still be a simple polygon but with edges that are so close to each other that will almost intersect. If a kernel with exact constructions is used, the offsetting algorithm can guarantee that the output contains only simple polygons. However, if inexact constructions are used the roundoff in the coordinates of the output points will cause parallel edges that *almost* collapse-but not so-to become really collinear or even cross each other.

Thus, it is neccessary to use a kernel with exact constructions if offset polygons must be simple, yet computing a straight skeleton using that kernel is very slow, much more than computing the offset polygons. To help with this, it is possible to construct the straight skeleton using the recommended kernel *Exact_predicates_inexact_constructions_kernel*, then convert the skeleton to a different kernel via the function *convert_straight_skeleton_2* and input the converted skeleton to the offsetting functions.

All the offsetting functions that take polygons as input (and create the straight skeleton under the hood) apply that optimization automatically: that is, the output polygons are defined over the same kernel of the input polygons, whatever that is, yet the straight skeleton is constructed with the faster recommended kernel and converted if necessary.

Notice how some of the examples above use *Exact_predicates_exact_constructions_kernel*. In all cases, the straight skeleton is constructed using *Exact_predicates_inexact_constructions_kernel*.

advanced

23.3.5 Low level API

All the high level functions described above are just wrappers around the low level API described here. This low level API is richer and provides options and configurations not covered by any of those functions.

The straight skeleton construction algorithm is encapsulated in the class *Straight_skeleton_builder_2*<*Gt,Ss,Visitor*> which is parameterized on a geometric traits (class *Straight_skeleton_builder_traits*<*Kernel*>) and the Straight Skeleton class (Ss).

The offset contours construction algorithm is encapsulated in the class *Polygon_offset_builder_2*<*Ss,Gt,Container*> which is parameterized on the Straight Skeleton class (Ss), a geometric traits (class *Polygon_offset_builder_traits*<*Kernel*>) and a container type where the resulting offset polygons are generated.

To construct the straight skeleton of a polygon with holes the user must:

(1) Instantiate the straight skeleton builder.

(2) Enter one contour at a time, starting from the outer contour, via the method *enter_contour*. The input polygon with holes must be non-degenerate, strictly-simple and counter-clockwise oriented (see the definitions at the beginning of this chapter). Collinear edges are allowed. The insertion order of each hole is unimportant but the outer contour must be entered first.

(3) Call *construct_skeleton* once *all* the contours have been entered. You cannot enter another contour once the skeleton has been constructed.

To construct a set of inward offset contours the user must:

(1) Construct the straight skeleton of the source polygon with holes.

(2) Instantiate the polygon offset builder passing in the straight skeleton as a parameter.

(3) Call *construct_offset_contours* passing the desired offset distance and an output iterator that can store a *boost::shared_ptr* of *Container* instances into a resulting sequence (typically, a back insertion iterator)

Each element in the resulting sequence is an *offset contour*, given by a *boost::shared_ptr* holding a dynamically allocated instance of the Container type. Such a container can be any model of the *VertexContainer_2* concept, for example, a *CGAL::Polygon_2*, or just a *std::vector* of 2D points.

The resulting sequence of offset contours can contain both outer and inner contours. Each offset hole (inner offset contour) would logically belong in the interior of some of the outer offset contours. However, this algorithm returns a sequence of contours in arbitrary order and there is no indication whatsoever of the parental relationship between inner and outer contours.

On the other hand, each outer contour is counter-clockwise oriented while each hole is clockwise-oriented. And since offset contours do form simple polygons with holes, it is guaranteed that no hole will be inside another hole, no outer contour will be inside any other contour, and each hole will be inside exactly 1 outer contour.

Parental relationships are *not* automatically reconstructed by this algorithm because this relation is not directly given by the input polygon with holes and must be done as a post processing step. The function *arrange_offset_polygons_2* can be used to do that efficiently.

A user can reconstruct the parental relationships as a post processing operation by testing each inner contour (which is identified by being clockwise) against each outer contour (identified as being counter-clockwise) for insideness.

This algorithm requires exact predicates but not exact constructions Therefore, the *Exact_predicates_inexact_constructions_kernel* should be used.

```
#include<vector>
#include<iterator>
#include<iostream>
#include<iomanip>
#include<string>

#include<boost/shared_ptr.hpp>

#include<CGAL/Cartesian.h>
#include<CGAL/Polygon_2.h>
#include<CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include<CGAL/Straight_skeleton_builder_2.h>
#include<CGAL/Polygon_offset_builder_2.h>
#include<CGAL/compute_outer_frame_margin.h>

#include "print.h"

//
// This example illustrates how to use the CGAL Straight Skeleton package
// to construct an offset contour on the outside of a polygon
//

// This is the recommended kernel
typedef CGAL::Exact_predicates_inexact_constructions_kernel Kernel;

typedef Kernel::Point_2 Point_2;
typedef CGAL::Polygon_2<Kernel>    Contour;
```

```
typedef boost::shared_ptr<Contour> ContourPtr;
typedef std::vector<ContourPtr>    ContourSequence ;

typedef CGAL::Straight_skeleton_2<Kernel> Ss;

typedef Ss::Halfedge_iterator Halfedge_iterator;
typedef Ss::Halfedge_handle   Halfedge_handle;
typedef Ss::Vertex_handle     Vertex_handle;

typedef CGAL::Straight_skeleton_builder_traits_2<Kernel>
   SsBuilderTraits;
typedef CGAL::Straight_skeleton_builder_2<SsBuilderTraits,Ss> SsBuilder;

typedef CGAL::Polygon_offset_builder_traits_2<Kernel>
   OffsetBuilderTraits;
typedef CGAL::Polygon_offset_builder_2<Ss,OffsetBuilderTraits,Contour>
   OffsetBuilder;

int main()
{
  // A start-shaped polygon, oriented counter-clockwise as required for
   outer contours.
  Point_2 pts[] = { Point_2(-1,-1)
                  , Point_2(0,-12)
                  , Point_2(1,-1)
                  , Point_2(12,0)
                  , Point_2(1,1)
                  , Point_2(0,12)
                  , Point_2(-1,1)
                  , Point_2(-12,0)
                  } ;

  std::vector<Point_2> star(pts,pts+8);

  // We want an offset contour in the outside.
  // Since the package doesn't support that operation directly, we use
   the following trick:
  // (1) Place the polygon as a hole of a big outer frame.
  // (2) Construct the skeleton on the interior of that frame (with the
   polygon as a hole)
  // (3) Construc the offset contours
  // (4) Identify the offset contour that corresponds to the frame and
   remove it from the result


  double offset = 3 ; // The offset distance

  // First we need to determine the proper separation between the polygon
    and the frame.
  // We use this helper function provided in the package.
  boost::optional<double> margin =
   CGAL::compute_outer_frame_margin(star.begin(),star.end(),offset);

  // Proceed only if the margin was computed (an extremely sharp corner
```

```
  might cause overflow)
 if ( margin )
 {
   // Get the bbox of the polygon
   CGAL::Bbox_2 bbox = CGAL::bbox_2(star.begin(),star.end());

   // Compute the boundaries of the frame
   double fxmin = bbox.xmin() - *margin ;
   double fxmax = bbox.xmax() + *margin ;
   double fymin = bbox.ymin() - *margin ;
   double fymax = bbox.ymax() + *margin ;

   // Create the rectangular frame
   Point_2 frame[4]= { Point_2(fxmin,fymin)
                     , Point_2(fxmax,fymin)
                     , Point_2(fxmax,fymax)
                     , Point_2(fxmin,fymax)
                     } ;

   // Instantiate the skeleton builder
   SsBuilder ssb ;

   // Enter the frame
   ssb.enter_contour(frame,frame+4);

   // Enter the polygon as a hole of the frame (NOTE: as it is a hole we
    insert it in the opposite orientation)
   ssb.enter_contour(star.rbegin(),star.rend());

   // Construct the skeleton
   boost::shared_ptr<Ss> ss = ssb.construct_skeleton();

   // Proceed only if the skeleton was correctly constructed.
   if ( ss )
   {
     print_straight_skeleton(*ss);

     // Instantiate the container of offset contours
     ContourSequence offset_contours ;

     // Instantiate the offset builder with the skeleton
     OffsetBuilder ob(*ss);

     // Obtain the offset contours
     ob.construct_offset_contours(offset,
  std::back_inserter(offset_contours));

     // Locate the offset contour that corresponds to the frame
     // That must be the outmost offset contour, which in turn must be
  the one
     // with the largetst unsigned area.
     ContourSequence::iterator f = offset_contours.end();
     double lLargestArea = 0.0 ;
     for (ContourSequence::iterator i = offset_contours.begin(); i !=
```

```
    offset_contours.end(); ++ i  )
      {
        double lArea = CGAL_NTS abs( (*i)->area() ) ; //Take abs() as
    Polygon_2::area() is signed.
        if ( lArea > lLargestArea )
        {
          f = i ;
          lLargestArea = lArea ;
        }
      }

      // Remove the offset contour that corresponds to the frame.
      offset_contours.erase(f);

      print_polygons(offset_contours);
    }
  }

  return 0;
}
```

File: examples/Straight_skeleton_2/Low_level_API.cpp

23.3.6 Exterior Skeletons and Exterior Offset contours

This CGAL package can only construct the straight skeleton and offset contours in the *interior* of a polygon with holes. However, constructing exterior skeletons and exterior offsets is possible:

Say you have some polygon made of 1 outer contour C0 and 1 hole C1, and you need to obtain some exterior offset contours.
The interior region of a polygon with holes is connected while the exterior region is not: there is an unbounded region outside the outer contour, and one bounded region inside each hole. To construct an offset contour you need to construct an straight skeleton. Thus, to construct exterior offset contours for a polygon with holes, you need to construct, *separately*, the exterior skeleton of the outer contour and the interior skeleton of each hole.
Constructing the interior skeleton of a hole is directly supported by this CGAL package; you just need to input the hole's vertices in reversed order as if it were an outer contour.
Constructing the exterior skeleton of the outer contour is possible by means of the following trick: place the contour as a hole of a big rectangle (call it *frame*). If the frame is sufficiently separated from the contour, the resulting skeleton will be practically equivalent to a *real* exterior skeleton.
To construct exterior offset contours in the inside of each hole you just use the skeleton constructed in the interior, and, if required, revert the orientation of each resulting offset contour.
Constructing exterior offset contours in the outside of the outer contour is just a little bit more involved: Since the contour is placed as a hole of a frame, you will always obtain 2 offset contours for any given distance; one is the offseted frame and the other is the offseted contour. Thus, from the resulting offset contour sequence, you always need to discard the offsetted frame, easily identified as the offset contour with the largest area.
It is necessary to place the frame sufficiently away from the contour. If it is not, it could occur that the outward offset contour collides and merges with the inward offset frame, resulting in 1 instead of 2 offset contours.
However, the proper separation between the contour and the frame is not directly given by the offset distance at which you want the offset contour. That distance must be at least the desired offset plus the largest euclidean distance between an offset vertex and its original.
This CGAL packages provides a helper function to compute the required separation: *compute_outer_frame_margin*

If you use this function to place the outer frame you are guaranteed to obtain an offset contour corresponding exclusively to the frame, which you can always identify as the one with the largest area and which you can simple remove from the result (to keep just the relevant outer contours).

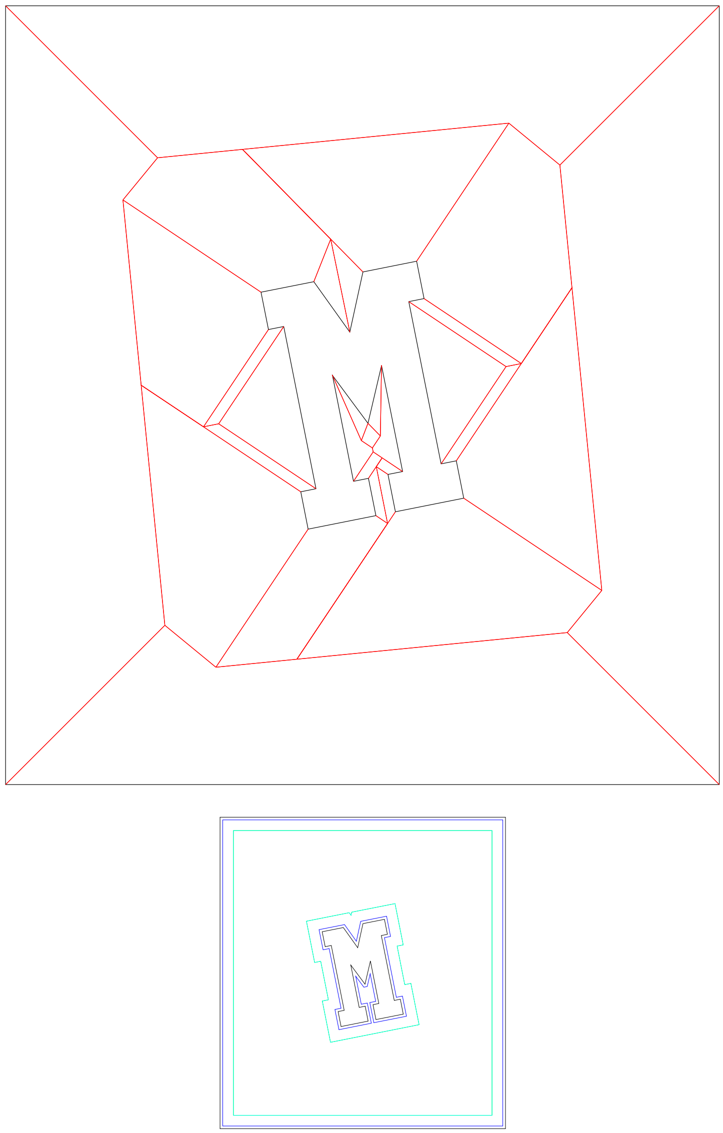


Figure 23.7: Exterior skeleton obtained using a frame (left) and 2 sample exterior offset contours (right)

23.4 Straight Skeletons, Medial Axis and Voronoi Diagrams

The straight skeleton of a polygon is similar to the medial axis and the Voronoi diagram of a polygon in the way it partitions it; however, unlike the medial axis and Voronoi diagram, the bisectors are not equidistant to its defining edges but to the supporting lines of such edges. As a result, Straight Skeleton bisectors might not be located in the center of the polygon and so cannot be regarded as a proper Medial Axis in its geometrical meaning.

On the other hand, only reflex vertices (whose internal angle $> pi$) are the source of deviations of the bisectors from its center location. Therefore, for convex polygons, the straight skeleton, the medial axis and the Voronoi diagram are exactly equivalent, and, if a non-convex polygon contains only vertices of low reflexivity, the straight skeleton bisectors will be placed nearly equidistant to their defining edges, producing a straight skeleton pretty much alike a proper medial axis.

23.5 Usages of the Straight Skeletons

The most natural usage of straight skeletons is offsetting: growing and shrinking polygons (provided by this CGAL package).

Another usage, perhaps its very first, is roof design: The straight skeleton of a polygonal roof directly gives the layout of each tent. If each skeleton edge is lifted from the plane a height equal to its offset distance, the resulting roof is "correct" in that water will always fall down to the contour edges (roof border) regardless of were in the roof it falls. [LD03] gives an algorithm for roof design based on the straight skeleton.

Just like medial axes, 2D straight skeletons can also be used for 2D shape description and matching. Essentially, all the applications of image-based skeletonization (for which there is a vast literature) are also direct applications of the straight skeleton, specially since skeleton edges are simply straight line segments.

Consider the subgraph formed only by *inner bisectors* (that is, only the skeleton halfedges which are not incident upon a contour vertex). Call this subgraph a *skeleton axis*. Each node in the skeleton axis whose degree is $>= 3$ roots more than one skeleton tree. Each skeleton tree roughly corresponds to a region in the input topologically equivalent to a rectangle; that is, without branches. For example, a simple letter "H" would contain 2 higher degree nodes separating the skeleton axis in 5 trees; while the letter "@" would contain just 1 higher degree node separating the skeleton axis in 2 curly trees.

Since a skeleton edge is a 2D straight line, each branch in a skeleton tree is a polyline. Thus, the path-length of the tree can be directly computed. Furthermore, the polyline for a particular tree can be interpolated to obtain curve-related information.

Pruning each skeleton tree cutting off branches whose length is below some threshold; or smoothing a given branch, can be used to reconstruct the polygon without undesired details, or fit into a particular canonical shape.

Each skeleton edge in a skeleton branch is associated with 2 contour edges which are facing each other. If the polygon has a bottleneck (it almost touches itself), a search in the skeleton graph measuring the distance between each pair of contour edges will reveal the location of the bottleneck, allowing you to cut the shape in two. Likewise, if two shapes are too close to each other along some part of their boundaries (a near contact zone), a similar search in an exterior skeleton of the two shapes at once would reveal the parts of near contact, allowing you to stitch the shapes. These *cut and stitch* operations can be directly executed in the straight skeleton itself instead of the input polygon (because the straight skeleton contains a graph of the connected contour edges).

23.6 Straight Skeleton of a General Figure in the Plane

A straight skeleton can also be defined for a general multiply-connected planar directed straight-line graph [AA95] by considering all the edges as embedded in an unbounded region. The only difference is that in this case some faces will be only partially bounded.

The current version of this CGAL package can only construct the straight skeleton in the interior of a simple polygon with holes, that is it doesn't handle general polygonal figures in the plane.

2D Straight Skeleton and Polygon Offsetting Reference Manual

This chapter introduces the concepts and classes that correspond to the CGAL *2D Straight Skeleton* package. The packages provides an algorithm for the construction of the straight skeleton in the interior of a simple polygon with holes and an algorithm for the construction of inward offset contours based on the straight skeleton.

23.7 Classified Reference Pages

Concepts

Classes

23.8 Alphabetical List of Reference Pages

StraightSkeleton_2

Concept

Definition

The concept StraightSkeleton_2 describes the requirements for the data structure used to represent a straight skeleton. It refines the concept *HalfedgeDS* and adds additional requirements on the nested types *Vertex*, *Halfedge*, and *Face* of the halfedge data structure.

Refines

HalfedgeDS

Types

StraightSkeleton_2:: Vertex	A model of the *StraightSkeletonVertex_2* concept used to represent the vertices of the straight skeleton
StraightSkeleton_2:: Halfedge	A model of the *StraightSkeletonHalfedge_2* concept used to represent the halfedges of the straight skeleton
StraightSkeleton_2:: Face	Any model of the *StraightSkeletonFace_2* concept used to represent the faces of the straight skeleton

Has Models

CGAL::Straight_skeleton_2<Traits,Items,Alloc>.

This concept explicitly protects all the modifying operations of the base *HalfedgeDS* concept. Only the algorithm classes, or clients explicitly bypassing the protection mechanism, can modify a straight skeleton.

Concept

StraightSkeletonVertex_2

Definition

The concept StraightSkeletonVertex_2 describes the requirements for the vertex type of the *StraightSkeleton_2* concept. It is a refinement of the *HalfedgeDSVertex* concept with support for storage of the incident halfedge. The StraightSkeletonVertex_2 concept requires the geometric embedding to be a 2D point.

Refines

HalfedgeDSVertex

Types

StraightSkeletonVertex_2:: Point_2 — The type of the 2D point being the geometric embedding of the vertex

StraightSkeletonVertex_2:: FT — A model of the *SqrtFieldNumberType* concept representing the time of a vertex (an Euclidean distance)

StraightSkeletonVertex_2:: Halfedge_around_vertex_const_circulator
StraightSkeletonVertex_2:: Halfedge_around_vertex_circulator

The circulator type used to visit all the incident halfedges around a vertex

StraightSkeletonVertex_2:: Defining_contour_halfedge_const_circulator
StraightSkeletonVertex_2:: Defining_contour_halfedge_circulator

The circulator type used to visit all the defining contour halfedges of a vertex

Creation

StraightSkeletonVertex_2 v; — Default constructor

StraightSkeletonVertex_2 v(int id, Point_2 p); — Constructs a contour vertex with ID number *id*, at the point *p*

StraightSkeletonVertex_2 v(int id, Point_2 p, FT time);

Constructs a skeleton vertex with ID number *id*, at point *p* and time *time*.

Access Functions

int	*v.id() const*	The ID of the vertex.
Point_2	*v.point() const*	The vertex point.
FT	*v.time() const*	The time of the vertex: the distance from the vertex point to the lines supporting the defining contour edges
Halfedge_handle	*v.primary_bisector()*	

Halfedge_const_handle

v.primary_bisector() const

Returns the skeleton halfedge incident upon the vertex (called the *primary* bisector).

Halfedge_around_vertex_circulator

v.halfedge_around_vertex_begin()

Halfedge_around_vertex_const_circulator

v.halfedge_around_vertex_begin() const

Returns a bi-directional circulator pointing to one of the incident halfedges (which one is unspecified).
There will always be as many incident halfedges as the degree of the vertex.
If this is a *contour* vertex, its degree is exactly 3, and from the halfedges pointed to by the circulator, 2 are contour and 1 is a bisector.
If this is an *skeleton* vertex, its degree is at least 3 and all of the halfedges pointed to by the circulator are bisectors.
Each halfedge pointed to by this circulator is the one which is oriented towards the vertex (according to the geometric embedding).

Defining_contour_halfedge_circulator

v.defining_contour_halfedges_begin()

Defining_contour_halfedge_const_circulator

v.defining_contour_halfedges_begin() const

Returns a bi-directional circulator pointing to one of the defining contour halfedges of the vertex (which one is unspecified).
There will always be as many incident defining contour halfedges as the degree of the vertex.
Each halfedge pointed to by this circulator is the one having its left side facing inwards (which happens to be the contour halfedge for which *is_border()* is *false*).

Queries

bool	*v.is_contour() const*	Returns *true* iff this is a contour vertex.
bool	*v.is_skeleton() const*	Returns *true* iff this is a skeleton vertex.

Has Models

CGAL::Straight_skeleton_vertex_base_2<Refs,Point,FT>.

See Also

StraightSkeleton_2
StraightSkeletonHalfedge_2
CGAL::Straight_skeleton_vertex_base_2<Refs,Point,FT>
CGAL::Straight_skeleton_halfedge_base_2<Refs>

StraightSkeletonHalfedge_2

Concept

Definition

The concept StraightSkeletonHalfedge_2 describes the requirements for the halfedge type of the *StraightSkeleton_2* concept. It is a refinement of the *HalfedgeDSHalfedge* concept. The StraightSkeletonHalfedge_2 concept requires no geometric embedding at all. The only geometric embedding used by the Straight Skeleton Data Structure are the 2D points in the contour and skeleton vertices. However, for any halfedge, there is a 2D segment implicitly given by its *source* and *target* vertices.

Refines

HalfedgeDSHalfedge

Creation

StraightSkeletonHalfedge_2 h;	Default Constructor.
StraightSkeletonHalfedge_2 h(int id);	Constructs a halfedge with ID *id*. It is the links to other halfedges what determines if this is a contour edge, a contour-skeleton edge or an inner-skeleton edge.

Access Functions

Halfedge_handle	*h.defining_contour_edge()*	
Halfedge_const_handle	*h.defining_contour_edge() const*	If this is a bisector halfedge, returns a handle to the inward-facing (non-border) contour halfedge corresponding to the defining contour edge which is to its left; if this is a contour halfedge, returns a handle to itself if *is_border()* is *false*, or to its opposite if it is *true*.

Queries

bool	*h.is_bisector() const*	Returns *true* iff this is a bisector (or skeleton) halfedge (i.e. is not a contour halfedge).
bool	*h.is_inner_bisector() const*	Returns *true* iff this is a bisector and is inner (i.e. is not incident upon a contour vertex).

Has Models

CGAL::Straight_skeleton_halfedge_2<Refs>.

See Also

StraightSkeleton_2
StraightSkeletonHalfedge_2
CGAL::Straight_skeleton_vertex_base_2<Refs,Point,FT>
CGAL::Straight_skeleton_halfedge_base_2<Refs>

StraightSkeletonFace_2

Definition

The concept StraightSkeletonFace_2 describes the requirements for the face type of the *StraightSkeleton_2* concept. It is a refinement of the *HalfedgeDSFace* concept with support for storage of the incident halfedge.

Refines

HalfedgeDSFace

Creation

StraightSkeletonFace_2 f;	Default constructor
StraightSkeletonFace_2 f(int id);	Constructs a face with ID number *id*

Access Functions

int	*f.id() const*	The ID of the face.

Has Models

CGAL::Straight_skeleton_face_base_2<Refs>.

See Also

StraightSkeleton_2
CGAL::Straight_skeleton_face_base_2<Refs>
CGAL::Straight_skeleton_vertex_base_2<Refs,Point,FT>
CGAL::Straight_skeleton_halfedge_base_2<Refs>

Concept

Concept

StraightSkeletonItemsConverter_2

Definition

The concept StraightSkeletonItemsConverter_2 describes the requirements for items converter passed as the third template argument to the class *Straight_skeleton_converter_2<SrcSs,TgtSs,ItemsConverter>*. It converts the *HDS* items from one type of straight skeleton to another

Types

StraightSkeletonItemsConverter_2:: Source_vertex_const_handle

A constant handle to a model of the *StraightSkeletonVertex_2* concept used to represent the vertices of the straight skeleton of the source type.

StraightSkeletonItemsConverter_2:: Source_halfedge_const_handle

A constant handle to model of the *StraightSkeletonHalfedge_2* concept used to represent the halfedges of the straight skeleton of the source type.

StraightSkeletonItemsConverter_2:: Source_face_const_handle

A constant handle to model of the *StraightSkeletonFace_2* concept used to represent the faces of the straight skeleton of the source type.

StraightSkeletonItemsConverter_2:: Tarvet_vertex

A model of the *StraightSkeletonVertex_2* concept used to represent the vertices of the straight skeleton of the target type.

StraightSkeletonItemsConverter_2:: Target_halfedge

A model of the *StraightSkeletonHalfedge_2* concept used to represent the halfedges of the straight skeleton of the target type.

StraightSkeletonItemsConverter_2:: Target_face

Any model of the *StraightSkeletonFace_2* concept used to represent the faces of the straight skeleton of the target type.

Operations

Target_vertex *conv.operator()(Source_vertex_const_handle v) const*

returns a new vertex with the same data as *v* converted to the corresponding target types.

Target_halfedge	*conv.operator()(Source_halfedge_const_handle h) const*	
		returns a new halfedge with the same data as h converted to the corresponding target types.
Target_face	*conv.operator()(Source_face_const_handle f) const*	
		returns a new face with the same data as f converted to the corresponding target types.

Has Models

CGAL::Straight_skeleton_items_converter_2<TgtSs,SrcSs,NTConverter>

See Also

CGAL::Straight_skeleton_converter_2<SrcSs,TgtSs,ItemsCvt> . . . . page 1722

StraightSkeletonBuilder_2_Visitor

Definition

The concept StraightSkeletonBuilder_2_Visitor describes the requirements of the visitor class required by the algorithm class *Straight_skeleton_builder_2<Gt,Ss,Visitor>* in its third template parameter.

Types

StraightSkeletonBuilder_2_Visitor:: Halfedge_const_handle

A constant handle to a straight skeleton halfedge.

StraightSkeletonBuilder_2_Visitor:: Vertex_const_handle

A constant handle to a straight skeleton vertex.

Creation

Operations

void *v.on_contour_edge_entered(Halfedge_const_handle const&) const*

Called for each contour halfedge added to the skeleton).

void *v.on_initialization_started(std::size_t number_of_vertices) const*

Called before the initialization stage (when initial events are discovered) is started.

void *v.on_contour_vertex_processed(Vertex_const_handle const& v, bool is_reflex, bool is_degenerate) const*

Called after the events for contour vertex v have been discovered. *is_reflex* is true if this vertex has an internal angle $> PI$, or *is_degenerate* is true if the internal angle is $= PI$.

void *v.on_edge_event_created(Vertex_const_handle const& node0, Vertex_const_handle const& node1) const*

Called after an edge event for nodes $node0$ and $node1$ has been discovered and put on the queue for later processing.

void *v.on_split_event_created(Vertex_const_handle const& node) const*

Called after a slipt event for node *node* has been discovered and put on the queue for later processing.

void *v.on_pseudo_split_event_created(Vertex_const_handle const& node0, Vertex_const_handle const& node1) const*

Called after a pseudo slipt event for nodes $node0$ and $node1$ has been discovered and put on the queue for later processing.

void *v.on_initialization_finished() const*

Called after all initial events have been discovered.

void *v.on_propagation_started() const*

Called before the propagation stage (when events are poped off the queue and processed) is started.

void *v.on_anihiliation_event_processed(Vertex_const_handle const& node0, Vertex_const_handle const& node1) const*

Called after an anhiliation event for nodes *node*0 and *node*1 has been processed. A new skeleton edge between these nodes has been added.

void *v.on_edge_event_processed(Vertex_const_handle const& seed0, Vertex_const_handle const& seed1, Vertex_const_handle const& newnode) const*

Called after an edge for nodes *seed*0 and *seed*1 has been processed. Skeleton vertex *newnode* and edges from *node*0 to *newnode* and *node*1 to *newnode* has been added.

void *v.on_split_event_processed(Vertex_const_handle const& seed, Vertex_const_handle const& newnode0, Vertex_const_handle const& newnode1) const*

Called after a split event for node *seed* has been processed. Skeleton vertices *newnode*0 and *newnode*1 have been added. An skeleton edge from *seed* to *newnode*0 has been added. In the final skeleton, *newnode*1 is removed and only *newnode*0 remains.

void *v.on_pseudo_split_event_processed(Vertex_const_handle const& seed0, Vertex_const_handle const& seed1, Vertex_const_handle const& newnode0, Vertex_const_handle const& newnode1) const*

Called after a pseudo split event for nodes *seed*0 and *seed*1 has been processed. Skeleton vertices *newnode*0 and *newnode*1 have been added. Skeleton edges from *seed*0 to *newnode*0 and *seed*1 to *newnode*1 has been added.

void *v.on_vertex_processed(Vertex_const_handle const& v) const*

Called after vertex *v* has been marked as already processed.

void *v.on_propagation_finished() const*

Called after all events have been processed.

void *v.on_cleanup_started() const*

Called when the skeleton clean up (when multiple nodes are merged) is started.

void *v.on_cleanup_finished() const*

Called when clean up finished.

void *v.on_algorithm_finished(bool finished_ok) const*

Called when the algorithm terminated. *finished_ok* is false if it terminated before completion or the resulting skeleton was found to be invalid.

void *v.on_error(char const* msg) const*

Called whenever an error was detected. *msg* is whatever error message accompanies the error. This pointer can be *null*.

Has Models

CGAL::Dummy_straight_skeleton_builder_2_visitor<Ss>.

See Also

CGAL::Straight_skeleton_builder_2<Gt,Ss>

StraightSkeletonBuilderTraits_2

Definition

The concept StraightSkeletonBuilderTraits_2 describes the requirements for the geometric traits class required by the algorithm class *Straight_skeleton_builder_2<Gt,Ss>*.

Types

StraightSkeletonBuilderTraits_2:: Kernel	A model of the *Kernel* concept.
StraightSkeletonBuilderTraits_2:: FT	A *SqrtFieldNumberType* provided by the kernel. This type is used to represent the coordinates of the input points, of the skeleton nodes and as the event time stored in the skeleton nodes.
boost::tuple<FT,FT> Vertex;	A pair of (x,y) coordinates representing a 2D Cartesian point.
boost::tuple<Vertex,Vertex> Edge;	A pair of vertices representing an edge
boost::tuple<Edge,Edge,Edge> EdgeTriple;	A triple of edges representing an event

StraightSkeletonBuilderTraits_2:: Equal_2

A predicate object type being a model of the *Kernel::Equal_2* function object concept.

StraightSkeletonBuilderTraits_2:: Left_turn_2

A predicate object type being a model of the *Kernel::LeftTurn_2* function object concept.

StraightSkeletonBuilderTraits_2:: Collinear_2

A predicate object type being a model of the *Kernel::Collinear_2* function object concept.

StraightSkeletonBuilderTraits_2:: Do_ss_event_exist_2

A predicate object type.
Must provide *bool operator()(EdgeTriple const& et) const*, which determines if, given the 3 *oriented* lines defined by the 3 input edges (3 pair of points), there exist an Euclidean distance $t \geq 0$ for which the corresponding 3 *offset lines at t* (parallel lines at an Euclidean distance of t) intersect in a single point.

Precondition: each edge in the triple must properly define an oriented line, that is, such points cannot be coincident.

StraightSkeletonBuilderTraits_2:: Compare_ss_event_times_2

A predicate object type.
Must provide *Comparison_result operator()(EdgeTriple const& x, EdgeTriple const& y) const*, which compares the *times* for the events determined by the edge-triples *x* and *y*.
The time of an event given by an edge triple (which defines 3 oriented lines) is the Euclidean distance *t* at which the corresponding offset lines at *t* intersect in a single point.

Precondition: *x* and *y* must be edge-triples corresponding to events that actually exist (as determined by the predicate *Exist_sls_event_2*).

StraightSkeletonBuilderTraits_2:: Compare_ss_event_distance_to_seed_2

A predicate object type.
Must provide *Comparison_result operator()(Point_2 const& p, EdgeTriple const& x, EdgeTriple const& y)*, which compares the Euclidean distance of the points of event for *x* and *y* to the point *p*.
The point of an event given by an edge triple (which defines 3 oriented lines) is the intersection point of the 3 corresponding offset lines at the time *t* of the event.
It must also provide *Comparison_result operator(EdgeTriple const& seed, EdgeTriple const& x, EdgeTriple const& y)()* which makes the same comparison as the first overload but where the seed point is given implicitly as the point of event for *seed*.

Precondition: *seed*, *x* and *y* must be edge-triples corresponding to events that actually exist (as determined by the predicate *Exist_sls_event_2*).

StraightSkeletonBuilderTraits_2:: Is_ss_event_inside_offset_zone_2

A predicate object type.
Must provide *bool operator()(EdgeTriple const& e, EdgeTriple const& zone)*, which determines if the point of event for *e* is inside the *offset zone* defined by the 3 *oriented* lines given by *zone*.
An offset zone given by 3 *oriented* lines is the intersection of the halfplanes to the left of each oriented line.

Precondition: *e* must be an edge-triple corresponding to an event that actually exist (as determined by the predicate *Exist_sls_event_2*), and the 3 oriented lines given by *zone* must be well defined (no point-pair can have coincident points).

StraightSkeletonBuilderTraits_2:: Are_ss_events_simultaneous_2

A predicate object type.
Must provide *bool operator()(EdgeTriple const& x, EdgeTriple const& y)*, which determines if the events given by *x* and *y* are coincident in time and space; that is, both triples of offset lines intersect at the same point and at the same Euclidean distance from their sources.

Precondition: *x* and *y* must be edge-triples corresponding to events that actually exist (as determined by the predicate *Exist_sls_event_2*).

StraightSkeletonBuilderTraits_2:: Construct_ss_event_time_and_point_2

A construction object type.
Must provide *boost::tuple< boost::optional<FT>, boost::optional<Point_2> > operator()(EdgeTriple const& e)*, which given the 3 *oriented* lines defined by the 3 input edges (3 pair of points), returns the Euclidean distance $t \geq 0$ and intersection point at which the corresponding 3 *offset lines at t* intersect.
If the values cannot be computed, not even approximately (because of overflow for instance), an empty optional must be returned.

Precondition: *e* must be an edge-triple corresponding to an event that actually exist (as determined by the predicate *Exist_sls_event_2*).

StraightSkeletonBuilderTraits_2:: Construct_ss_vertex_2

A construction object type.
Must provide *Vertex operator()(Point_2 const& p)*, which given a *Point_2 p* returns a Vertex encapsulating the corresponding (x,y) pair of *Cartesian* coordinates.

StraightSkeletonBuilderTraits_2:: Construct_ss_edge_2

A construction object type.
Must provide *Edge operator()(Point_2 const& s, Point_2 const& t)*, which given source and target points *s* and *t* returns an Edge encapsulating the corresponding input segment (in *Cartesian* coordinates.)

StraightSkeletonBuilderTraits_2:: Construct_ss_triedge_2

A construction object type.
Must provide *Triedge operator()(Edge const& e0, Edge const& e1, Edge const& e2)*, which given the 3 edges that define an event, *e0*, *e1* and *e2*, returns a Triedge encapsulating them.

Has Models

CGAL::Straight_skeleton_builder_traits_2<K>.

See Also

CGAL::Straight_skeleton_builder_2<Gt,Ss>
CGAL::Straight_skeleton_builder_traits_2<K>

Concept

PolygonOffsetBuilderTraits_2

Definition

The concept PolygonOffsetBuilderTraits_2 describes the requirements for the geometric traits class required by the algorithm class *Polygon_offset_builder_2<Ss,Gt,Polygon_2>*.

Types

PolygonOffsetBuilderTraits_2:: Kernel	A model of the *Kernel* concept.
PolygonOffsetBuilderTraits_2:: FT	A *SqrtFieldNumberType* provided by the kernel. This type is used to represent the coordinates of the input points and to specify the desired offset distance.
PolygonOffsetBuilderTraits_2:: Point_2	A 2D point type
boost::tuple<FT,FT> Vertex;	A pair of (x,y) coordinates representing a 2D Cartesian point.
boost::tuple<Vertex,Vertex> Edge;	A pair of vertices representing an edge
boost::tuple<Edge,Edge,Edge> EdgeTriple;	A triple of edges representing an event

PolygonOffsetBuilderTraits_2:: Compare_offset_against_event_time_2

A predicate object type.
Must provide *Comparison_result operator()(FT d, EdgeTriple const& et) const*, which compares the Euclidean distance d with the event time for *et*. Such event time is the Euclidean distance at which the *offset lines* intersect in a single point. The source of such offset lines is given by the 3 *oriented* lines defined by the edge-triple *et*
Precondition: *et* must be an edge-triple corresponding to an event that actually exist (that is, there must exist an offset distance $t > 0$ at which the offset lines do intersect at a single point.

PolygonOffsetBuilderTraits_2:: Construct_offset_point_2

A construction object type.
Must provide *boost::optional<Point_2> operator()(FT t, Edge const& x, Edge const& y) const*, which constructs the point of intersection of the lines obtained by offsetting the oriented lines given by x and y an Euclidean distance t. If the point cannot be computed, not even approximately (because of overflow for instance), an empty optional must be returned.

Precondition: x and y must intersect in a single point

PolygonOffsetBuilderTraits_2:: Construct_ss_vertex_2

A construction object type.
Must provide *Vertex operator()(Point_2 const& p)*, which given a *Point_2 p* returns a Vertex encapsulating the corresponding (x,y) pair of Cartesian coordinates.

PolygonOffsetBuilderTraits_2:: Construct_ss_edge_2

A construction object type.
Must provide *Edge operator()(Point_2 const& s, Point_2 const& t)*, which given source and target points *s* and *t* returns an Edge encapsulating the corresponding input segment (in Cartesian coordinates.)

PolygonOffsetBuilderTraits_2:: Construct_ss_triedge_2

A construction object type.
Must provide *Triedge operator()(Edge const& e0, Edge const& e1, Edge const& e2)*, which given the 3 edges that define an event, *e0*, *e1* and *e2*, returns a Triedge encapsulating them.

Has Models

CGAL::Polygon_offset_builder_traits_2<K>.

See Also

CGAL::Polygon_offset_builder_2<Ss,Gt,Polygon_2>
CGAL::Polygon_offset_builder_traits_2<K>

Concept

VertexContainer_2

Introduction

A model for the VertexContainer_2 concept defines the requirements for a resizable container of 2D points. It is used to output the offset polygons generated by the *Polygon_offset_builder_2<Ssds,Gt,Container>* class.

Types

VertexContainer_2:: Point_2	A 2D point type used to represent a vertex. Must be a model of the *Kernel::Point_2* concept
VertexContainer_2:: size_type	A unsigned integral type that can represent the number of vertices in the container.

Creation

VertexContainer_2 c;		Default constructor
size_type	*c.size() const*	Returns the number of vertices in the container.
void	*c.push_back(Point_2 v) const*	Adds the vertex v to the container, resizing its capacity if required.

Has Models

CGAL::Polygon_2

Any *standard BackInsertionSequence*, such as *vector*, *list* or *deque*, with a *value_type* being a model of the *Kernel::Point_2* concept.

CGAL::create_interior_straight_skeleton_2

Function

Definition

The function *create_interior_straight_skeleton_2* creates a straight skeleton in the interior of a 2D polygon with holes.

#include <CGAL/create_straight_skeleton_2.h>

```
template<class PointIterator, class HoleIterator, class K>
boost::shared_ptr< Straight_skeleton_2<K> >
        create_interior_straight_skeleton_2( PointIterator outer_contour_vertices_begin,
                                             PointIterator outer_contour_vertices_end,
                                             HoleIterator holes_begin,
                                             HoleIterator holes_end,
                                             K k = Exact_predicates_inexact_constructions_kernel)
```

returns a new *Straight_skeleton_2<K>* in the interior of the 2D polygon with holes whose outer boundary is given by the point sequence *[outer_contour_vertices_begin,outer_contour_vertices_end]* and its holes given by *[holes_begin,holes_end]*.

```
template<class PointIterator, class K>
boost::shared_ptr< Straight_skeleton_2<K> >
        create_interior_straight_skeleton_2( PointIterator outer_contour_vertices_begin,
                                             PointIterator outer_contour_vertices_end,
                                             K k = Exact_predicates_inexact_constructions_kernel)
```

returns a new *Straight_skeleton_2<K>* in the interior of the 2D polygon whose outer boundary is given by the point sequence *[outer_contour_vertices_begin,outer_contour_vertices_end]*.

```
template<class Polygon, class K>
boost::shared_ptr< Straight_skeleton_2<K> >
        create_interior_straight_skeleton_2( Polygon outer_contour,
                                             K k = Exact_predicates_inexact_constructions_kernel)
```

returns a new *Straight_skeleton_2<K>* in the interior of the 2D polygon *outer_contour*.

1. *K* is any CGAL kernel.

2. *PointIterator::value_type* is equivalent to *K2::Point_2*. A cartesian converter is used to convert from *K2::Point_2* to *K::Point_2*

3. *HoleIterator::value_type* and *Polygon* are *Polygon_2<K>* or a standard container of *K2::Point_2* elements

#include *<CGAL/create_straight_skeleton_from_polygon_with_holes_2.h>*

template<class K>
boost::shared_ptr< Straight_skeleton_2<K> >

create_interior_straight_skeleton_2(Polygon_with_holes<K> poly_with_holes,
K k = Exact_predicates_inexact_constructions_kernel)

returns a new *Straight_skeleton_2<K>* in the interior of the 2D polygon with holes *poly_with_holes*.

See Also

create_exterior_straight_skeleton_2
Straight_skeleton_builder_2<Gt,Ss>

CGAL::create_exterior_straight_skeleton_2

Function

Definition

The function *create_exterior_straight_skeleton_2* creates a straight skeleton in the exterior of a 2D polygon with holes.

#include <CGAL/create_straight_skeleton_2.h>

template<class FT, class PointIterator, class K>
boost::shared_ptr< Straight_skeleton_2<K> >

create_exterior_straight_skeleton_2(FT max_offset,
PointIterator vertices_begin,
PointIterator vertices_end,
K k = Exact_predicates_inexact_constructions_kernel)

returns a new *Straight_skeleton_2<K>* in the *limited exterior* of the 2D polygon P given by the point sequence *[vertices_begin,vertices_end]*. The skeleton in the *limited exterior* of P is the skeleton in the interior of a polygon Q with P as its hole and a rectangular frame F as outer boundary. The outer boundary F is constructed by enlarging the bounding box of P a distance d. d is a margin sufficiently large to allow an outer offset at dinstance *max_offset* to be obtained from this exterior skeleton, as computed by the function *compute_outer_frame_margin*

template<class FT, class Polygon, class K>
boost::shared_ptr< Straight_skeleton_2<K> >

create_exterior_straight_skeleton_2(FT max_offset,
Polygon P,
K k = Exact_predicates_inexact_constructions_kernel)

returns a new *Straight_skeleton_2<K>* in the *limited exterior* of the 2D polygon P . The skeleton in the *limited exterior* of P is the skeleton in the interior of a polygon Q with P as its hole and a rectangular frame F as outer boundary. The outer boundary F is constructed by enlarging the bounding box of P a distance d. d is a margin sufficiently large to allow an outer offset at dinstance *max_offset* to be obtained from this exterior skeleton, as computed by the function *compute_outer_frame_margin*

Requirements

1. *K* is any CGAL kernel.
2. *PointIterator::value_type* is equivalent to *K2::Point_2*. A cartesian converter is used to convert from *K2::Point_2* to *K::Point_2*
3. *Polygon* is *Polygon_2<K>* or a standard container of *K2::Point_2* elements

See Also

create_interior_straight_skeleton_2
Straight_skeleton_builder_2<Gt,Ss,Visitor>

CGAL::Straight_skeleton_2<Traits,Items,Alloc>

Definition

The class *Straight_skeleton_2<Traits,Items,Alloc>* provides a model for the *StraightSkeleton_2* concept which is the class type used to represent a straight skeleton.
It inherits from *HalfedgeDS_vector<Traits,Items,Alloc>*

#include <CGAL/Straight_skeleton_2.h>

Is Model for the Concepts

StraightSkeleton_2
DefaultConstructible
CopyConstructible
Assignable

See Also

StraightSkeletonVertex_2
StraightSkeletonHalfedge_2
StraightSkeleton_2

The only purpose of this class is to protect all the modifying operations in a *HalfedgeDS*. Normal users should not modify a straight skeleton. If an advanced user needs to get access to the modifying operations, it must call the required methods through the *::Base* class.

Class

CGAL::Straight_skeleton_vertex_base_2<Refs,Point,FT>

Definition

The class *Straight_skeleton_vertex_base_2<Refs,Point,FT>* provides a model for the *StraightSkeletonVertex_2* concept which is the vertex type required by the *StraightSkeleton_2* concept. The class *Straight_skeleton_vertex_base_2<Refs,Point,FT>* has three template arguments: the first is the model of the *StraightSkeleton_2* concept (the vertex container), the second is a Point type, and the third is a model of the *SqrtFieldNumberType*, which is the numeric type used to represent the time of a vertex (a Euclidean distance).

This class can be used as a base class allowing users of the straight skeleton data structure to decorate a vertex with additional data. The concrete vertex class must be given in the *HalfedgeDSItems* template parameter of the instantiation of the *HalfedgeDS_default* class used as the model for the *Straight_skeleton_2* concept.

#include <CGAL/Straight_skeleton_vertex_base_2.h>

Is Model for the Concepts

StraightSkeletonVertex_2
DefaultConstructible
CopyConstructible
Assignable

See Also

StraightSkeletonVertex_2
StraightSkeletonHalfedge_2
StraightSkeleton_2
CGAL::Straight_skeleton_halfedge_base_2<Refs>

CGAL::Straight_skeleton_halfedge_base_2<Refs>

Class

Definition

The class *Straight_skeleton_halfedge_base_2<Refs>* provides a model for the *StraightSkeletonHalfedge_2* concept which is the halfedge type required by the *StraightSkeleton_2* concept. The class *Straight_skeleton_halfedge_base_2<Refs>* has only one template argument: a model of the *StraightSkeleton_2* concept (the halfedge container).

This class can be used as a base class allowing users of the straight skeleton data structure to decorate a halfedge with additional data. The concrete halfedge class must be given in the *HalfedgeDSItems* template parameter of the instantiation of the *HalfedgeDS_default* class used as the model for the *Straight_skeleton_2* concept.

#include <CGAL/Straight_skeleton_halfedge_base_2.h>

Is Model for the Concepts

StraightSkeletonHalfedge_2
DefaultConstructible
CopyConstructible
Assignable

See Also

StraightSkeletonHalfedge_2
StraightSkeletonVertex_2
StraightSkeleton_2
CGAL::Straight_skeleton_vertex_base_2<Refs,Point,FT>

Class

CGAL::Straight_skeleton_face_base_2<Refs>

Definition

The class *Straight_skeleton_face_base_2<Refs>* provides a model for the *StraightSkeletonFace_2* concept which is the face type required by the *StraightSkeleton_2* concept. The class *Straight_skeleton_face_base_2<Refs>* has only one template argument: the model of the *StraightSkeleton_2* concept (the face container).

This class can be used as a base class allowing users of the straight skeleton data structure to decorate a face with additional data. The concrete face class must be given in the *HalfedgeDSItems* template parameter of the instantiation of the *HalfedgeDS_default* class used as the model for the *Straight_skeleton_2* concept.

#include <CGAL/Straight_skeleton_face_base_2.h>

Is Model for the Concepts

StraightSkeletonFace_2
DefaultConstructible
CopyConstructible
Assignable

See Also

StraightSkeletonFace_2
StraightSkeletonVertex_2
StraightSkeletonHalfedge_2
StraightSkeleton_2
CGAL::Straight_skeleton_vertex_base_2<Refs>
CGAL::Straight_skeleton_halfedge_base_2<Refs>

CGAL::Straight_skeleton_items_converter_2<SrcSs, TgtSs, NTConverter>

Class

Definition

*Straight_skeleton_items_converter_2<SrcSs, TgtSs, NTConverter>*is a model of the *StraightSkeletonItemsConverter_2* concept

The first and second template parameters are the target and source straight skeleton types, resp.

The third template parameter *NTConverter* is a function object that must provide *TgtSs:Traits::FT operator()(SrcSs::Traits::FT n)* that converts *n* to an *TgtSs::Traits::FT* which has the same value. The default value of this parameter is *CGAL::NT_converter<SrcSs::Traits::FT, TgtSs::Traits::FT>*.

Is Model for the Concepts

StraightSkeletonItemsConverter_2
DefaultConstructible
CopyConstructible
Assignable

#include <CGAL/Straight_skeleton_converter_2.h>

Creation

Straight_skeleton_items_converter_2<SrcSs, TgtSs, NTConverter> conv;

Default constructor.

See Also

CGAL::Straight_skeleton_2<Traits,Items,Alloc> . . . . page 1717
CGAL::Straight_skeleton_converter_2<SrcSs,TgtSs,ItemsCvt> . . . . page 1722
CGAL::Straight_skeleton_builder_2<Gt,Ss,Visitor> . . . . page 1726

Class

CGAL::Straight_skeleton_converter_2<SrcSs,TgtSs,ItemsCvt>

Definition

The class *Straight_skeleton_converter_2<SrcSs,TgtSs,ItemsCvt>* converts a straight skeleton instantiated using certain traits into another straight skeleton instantiated using a different traits.

The first and second template parameters are the source and target straight skeleton types, resp.

The third template parameter *ItemsConverter* is a function object that must model the concept *StraightSkeletonItemsConverter_2*. The default value of this parameter is *CGAL::Straight_skeleton_items_converter_2<SrcSs,TgtSs>*.

This conversion can be used to produce a straight skeleton with the fast *Exact_predicates_inexact_constructions_kernel* but then input that skeleton into *Polygon_offset_builder<Ss,Gt,Container>* instantiated with the slower *Exact_predicates_exact_constructions_kernel* obtaining only simple offset polygons without paying the runtime overhead of exact constructions for the straight skeleton itself.

Types

SrcSs	*Source_skeleton;*	The *SrcSs* template parameter corresponding to the source straight skeleton.
TgtSs	*Target_skeleton;*	The *TgtSs* template parameter corresponding to the target straight skeleton.
ItemsCvt	*Items_converter;*	The *ItemsCvt* template parameter corresponding to the items converter.

#include <CGAL/Straight_skeleton_converter_2.h>

Creation

Straight_skeleton_converter_2<SrcSs,TgtSs,ItemsCvt> c(Items_converter c);

Default constructor

Operations

boost::shared_ptr<Target_skeleton>

c.operator()(Source_skeleton const& s) const

returns a new straight skeleton data structure with the same combinatorial and geometric data as *s* using the items converter to convert the geometric embeeding to the types of the target traits.

See Also

StraightSkeletonItemsConverter_2
Straight_skeleton_items_converter_2<SrcSs,TgtSs,NTConverter>
convert_straight_skeleton_2<SrcSs,TgtSs,NTConverter>

Function

CGAL::convert_straight_skeleton_2<TgtSs,SrcSs,ItemsCvt>

Definition

The function *convert_straight_skeleton_2<TgtSs,SrcSs,ItemsCvt>* converts a straight skeleton instantiated using certain traits into another straight skeleton instantiated using a different traits.

#include <CGAL/Straight_skeleton_converter_2.h>

boost::shared_ptr<Target_skeleton, Source_skeleton, Items_converter>

convert_straight_skeleton_2(Source_skeleton s,
Items_converted ic = Items_converter())

returns a new straight skeleton data structure with the same combinatorial and geometric data as *s* using the items converter *ic* to convert the geometric embedding to the types of the target skeleton.
The first template parameter is the type of the target straight skeleton. This template parameter is not bound to a function argument so it must be explicitly specified.

See Also

StraightSkeletonItemsConverter_2
Straight_skeleton_items_converter_2<SrcSs,TgtSs,NTConverter>
Straight_skeleton_converter_2<SrcSs,TgtSs,ItemsConverter>

CGAL::Straight_skeleton_builder_traits_2<Kernel>

Class

Definition

The class *Straight_skeleton_builder_traits_2<Kernel>* provides a model for the *StraightSkeletonBuilderTraits_2* concept which is the traits class required by the *Straight_skeleton_builder_2* algorithm class. The class *Straight_skeleton_builder_traits_2<Kernel>* has one template argument: a 2D CGAL Kernel. This parameter must be a model for the *Kernel* concept, such as the *Exact_predicates_inexact_constructions_kernel*, which is the recommended one.
It is unspecified which subset of the kernel is used into the output sequence and the returned iterator will be equal to *out*.

For any given input polygon, it in this traits class (and by extension in the builder class). This is to avoid restricting the choices in the implementation.

#include <CGAL/Straight_skeleton_builder_traits_2.h>

Is Model for the Concepts

StraightSkeletonBuilderTraits_2
DefaultConstructible
CopyConstructible

See Also

CGAL::Straight_skeleton_builder_2<Gt,Ssds>

Class

CGAL::Straight_skeleton_builder_2<Gt,Ss,Visitor>

Definition

The class *Straight_skeleton_builder_2<Gt,Ss,Visitor>* encapsulates the construction of the 2D straight skeleton in the interior of a polygon with holes. Its first template parameter, *Gt*, must be a model of the *StraightSkeletonBuilderTraits_2* concept, its second template parameter, *Ss*, must be a model of the *StraightSkeleton_2* concept, and its third template paramter, *Visitor*, must be a model of the *StraightSkeletonBuilder_2_Visitor* concept. If this last parameter is ommitted, the class *CGAL::Dummy_straight_skeleton_builder_2_visitor<Skel>* is used.

#include <CGAL/Straight_skeleton_builder_2.h>

Types

Straight_skeleton_builder_2<Gt,Ss,Visitor>:: Gt

The geometric traits (first template parameter)

Straight_skeleton_builder_2<Gt,Ss,Visitor>:: Ss

The straight skeleton (second template parameter)

Straight_skeleton_builder_2<Gt,Ss,Visitor>:: Point_2

The 2D point type as defined by the geometric traits

Creation

Straight_skeleton_builder_2<Gt,Ss,Visitor> b; Default constructs the builder class.

Methods

template<class InputPointIterator>

Straight_skeleton_builder_2&

b.enter_contour(InputPointIterator aBegin, InputPointIterator aEnd)

Defines the *contours* that form the *non-degenerate strictly-simple polygon with holes* whose *straight skeleton* is to be built.
Each contour must be input in turn starting with the *outer contour* and following with the holes (if any). The order of the holes is unimportant but the outer contour must be entered first. The outer contour must be oriented counter-clockwise while holes must be oriented clockwise.
It is an error to enter more than one outer contour or to enter a hole which is not inside the outer contour or inside another hole. It is also an error to enter a contour which crosses or touches any one another. It is possible however to enter a contour that touches itself in such a way that its interior region is still well defined and singly-connected (see the User Manual for examples).
The sequence [aBegin,aEnd) must iterate over each 2D point that corresponds to a vertex of the contour being entered. Vertices cannot be coincident (except consecutively since the method simply skip consecutive coincident vertices). Consecutive collinear edges are allowed.
InputPointIterator must be an *InputIterator* whose *value_type* is *Point_2*.

boost::shared_ptr<Ss>

b.construct_skeleton()

Constructs and returns the 2D straight skeleton in the interior of the polygon with holes as defined by the contours entered first by calling *enter_contour*. All the contours of the polygon with holes must be entered before calling *construct_skeleton*.
After *construct_skeleton* completes, you cannot enter more contours and/or call *construct_skeleton()* again. If you need another straight skeleton for another polygon you must instantiate and use another builder.
The result is a dynamically allocated instance of the *Ss* class, wrapped in a *boost::shared_ptr*.
If the construction process fails for whatever reason (such as a nearly-degenerate vertex whose internal or external angle is almost zero), the return value will be *null*, represented by a default constructed *shared_ptr*.
The algorithm automatically checks the consistency of the result, thus, if it is not *null*, it is guaranteed to be valid.

Algorithm

The implemented algorithm is closely based on [FO98] with the addition of *vertex events* as described in [EE98].

It simulates a grassfire propagation of moving polygon edges as they move inward at constant and equal speed. That is, the continuous inward offsetting of the polygon.

Since edges move at equal speed their movement can be characterized in a simpler setup as the movement of vertices. Vertices move along the angular bisector of adjacent edges.
The trace of a moving vertex is described by the algorithm as a *bisector*. Every position along a bisector corresponds to the vertex between two offset (moved) edges. Since edges move at constant speed, every position along a bisector also corresponds to the distance those two edges moved so far.
From the perspective of a dynamic system of moving edges, such a distance can be regarded as an *instant* (in time). Therefore, every distinct position along a bisector corresponds to a distinct instant in the offsetting process.
As they move inward, edges can expand or contract w.r.t to the endpoints sharing a vertex. If a vertex has an internal angle $< pi$, its incident edges will contract but if its internal angle $> pi$, they will expand. The movement of the edges, along with their extent change, result in collisions between non-adjacent edges. These collisions are called *events*, and they occur when the colliding edges have moved a certain distance, that is, at certain *instants*.
If non-consecutive edges *E(j),E(k)* move while edge *E(i)* contracts, they can collide at the point when *E(i)* shrinks to nothing (that is, the three edges might meet at a certain offset). This introduces a *topological change* in the polygon: Edges *E(j),E(k)* are now adjacent, edge *E(i)* disappears, and a new vertex appears. This topological change is called an *edge event*.
Similarly, consecutive expanding edges *E(i),E(i+1)* sharing a reflex vertex (internal angle $>= pi$) might collide with any edge *E(j)* on the rest of the same connected component of the polygon boundary (even far away from the initial edge's position). This also introduces a topological change: *E(j)* gets split in two edges and the connected component having *E(i),E(i+1) and E(j)* is split in two unconnected parts: one having *E(i)* and the corresponding subsegment of *E(j)* and the other with *E(i+1)* and the rest of *E(j)*. This is called a *split event*.
If a reflex vertex hits not an edge *E(j)* but another reflex vertex *E(j),E(j+1)*, and vice-versa (the reflex vertex V(j) hits V(i)), there is no actual split and the two unconnected parts have *E(i),E(j)* and *E(i+1),E(j+1)* (or *E(i),E(j+1)* and *E(i+1),E(j)*). This topological change is called a *vertex event*. Although similar to a split event in the sense that two new unconnected contours emerge introducing two new contour vertices, in the case of a vertex event one of the new contour vertices might be reflex; that is, a vertex event *may* result in one of the offset polygons having a *reflex* contour vertex which was not in the original polygon.

Edges movement is described by vertices movement, and these by bisectors. Therefore, the collision between edges *E(j),E(i),E(k)* (all in the same connected component) occurs when the moving vertices *E(j)->E(i)* and *E(i)->E(k)* meet ; that is, when the two bisectors describing the moving vertices *intersect* (Note: as the edges move inward and events occur, a vertex between edges A and B might exist even if A and B are not consecutive; that is, j and k are not necessarily i-1 and i+1 respectively, although initially they are).
Similarly, the collision between *E(i),E(i+1)* with *E(j)* (all in the same connected component) occurs when the bisector corresponding to the moving vertex *E(i)->E(i+1)* hits the moving edge *E(j)*.
Since each event changes the topology of the moving polygon, it is not possible or practical to foresee all events at once. Rather, the algorithm starts by estimating an initial set of potential events and from there it computes one next event at a time based on the previous one. The chaining of events is governed by their relative instants: events that occur first are processed first.
An initial set of *potential* split events is first computed independently (the computation of a potential split event is based solely on a reflex vertex and all other edges in the same connected component); and an initial set of *potential* edge events between initially consecutive bisectors is first computed independently (based solely on each bisector pair under consideration).
Events occur at certain instants and the algorithm must be able to order them accordingly. The correctness of the algorithm is uniquely and directly governed by the correct computation and ordering of the events. Any potential event might no longer be applicable after the topological change introduced by a prior event.
A grassfire propagation picks the next unprocessed event (starting from the first) and if it is still applicable processes it. Processing an event involves connecting edges, adding a new skeleton vertex (which corresponds the a contour vertex of the offset polygon) and calculating one new potential future event (which can be either an edge event or a split event -because of a prior vertex event-), based on the topological change just introduced. The propagation finishes when there are no new future events.

See Also

StraightSkeletonBuilderTraits_2
StraightSkeletonVertex_2
StraightSkeletonHalfedge_2
StraightSkeleton_2
CGAL::Straight_skeleton_builder_traits_2<K>
CGAL::Straight_skeleton_vertex_base_2<Refs,P,FT>
CGAL::Straight_skeleton_halfedge_base_2<Refs>
CGAL::Straight_skeleton_2<Traits,Items,Alloc>

Class

CGAL::Dummy_straight_skeleton_builder_2_visitor<Ss>

Definition

The class *Dummy_straight_skeleton_builder_2_visitor<Ss>* provides a model for the *StraightSkeletonBuilder_2_Visitor* concept which is the visitor type required by the *StraightSkeletonBuilder_2* concept. The class *Dummy_straight_skeleton_builder_2_visitor<Ss>* has only one template argument: a model of the *StraightSkeleton_2* concept.

This class is the default visitor parameter of the straight skeleton builder. All its methods are empty.

#include <CGAL/Straight_skeleton_builder_2.h>

Is Model for the Concepts

StraightSkeletonBuilder_2_Visitor
DefaultConstructible

See Also

CGAL::Straight_skeleton_builder_2<Gt,Ss,Visitor>

CGAL::create_offset_polygons_2

Function

Definition

The function *create_offset_polygons_2* creates a straight skeleton in the interior of a 2D polygon with holes.

#include <CGAL/create_straight_skeleton_2.h>

template<class Polygon, class FT, class Straight_skeleton, class K>
std::vector< boost::shared_ptr<Polygon> >

create_offset_polygons_2(FT offset,
Straight_skeleton s,
,
K k = Exact_predicates_inexact_constructions_kernel)

returns a container with all the offset polygons at distance $offset$ obtained from the straight skeleton s.

1. K is any CGAL kernel.
2. FT is any number type implicitly convertible to $K::FT$.
3. *Straight_skeleton* is *Straight_skeleton_2<K2>*. If $K! = K2$ the straight skeleton is converted to *Straight_skeleton_2<K>*.
4. *Polygon* is a model of *VertexContainer_2*. If this first template parameter is omitted, *Polygon_2* is used.

See Also

create_interior_skeleton_2
create_exterior_skeleton_2
create_interior_skeleton_and_offset_polygons_2
create_exterior_skeleton_and_offset_polygons_2

Function

CGAL::arrange_offset_polygons_2

Definition

The function *arrange_offset_polygons_2* arranges the sequence of *Polygon_2* objects obtained by *create_offset_polygons_2* into *Polygon_with_holes_2* objects by determining geometric parent-hole relationships using a simple algorithm based on the particular characteristics of offset polygons. This function should not be used to arrange arbitrary polygons into polygons with holes unless they meet the requirements specified below.

#include <CGAL/arrange_offset_polygons_2.h>

```
template<class K, class InputPolygonPtrIterator, class OutputPolygonWithHolesPtrIterator>
void arrange_offset_polygons_2( InputPolygonPtrIterator begin,
                                InputPolygonPtrIterator end,
                                OutputPolygonWithHolesPtrIterator out,
                                K k)
```

determines parent-hole relationships among the polygons given by *[begin,end]* creating *boost::shared_ptr< Polygon_with_holes_2<K> >* objects added to the output sequence given *out*. A *CLOCKWISE* oriented polygon *H* is a hole of a *COUNTERCLOCKWISE* polygon *P*, iff at least one vertex of *H* is *ON_BOUNDED_SIDE* of *P*.

Requirements

1. *K* is any CGAL kernel.
2. *InputPolygonPtrIterator* is an input iterator whose *value_type* is a *smart ptr* (such as *boost::shared_ptr*) whose *element_type* is *Polygon_2<K>*.
3. *InputPolygonPtrIterator* is an output iterator whose *value_type* is a *smart ptr* (such as *boost::shared_ptr*) whose *element_type* is *Polygon_with_holes_2<K>*.
4. The input polygons must be simple.
5. The set of input polygons are unique and interior disjoint. That is, given distinct polygons *P* and *Q*, there are vertices of *P* which are not on the boundary of *Q* and are all on the bounded or unbounded side of *Q* (but not both).

See Also

create_exterior_straight_skeleton_2
Straight_skeleton_builder_2<Gt,Ss>

CGAL::create_interior_skeleton_and_offset_polygons_2

Definition

The function *create_interior_skeleton_and_offset_polygons_2* creates inner offset polygons directly from an input polygon via a temporarily created straight skeleton.

#include <CGAL/create_offset_polygons_2.h>

```
template<class FT, class Polygon, class HoleIterator, class OffsettingK, class SkeletonK>
std::vector< boost::shared_ptr<Polygon> >
        create_interior_skeleton_and_offset_polygons_2( FT offset,
                                                        Polygon outer_boundary,
                                                        HoleIterator holes_begin,
                                                        HoleIterator holes_end,
                                                        OffsettingK ofk = CGAL::Exact_predicates_inexact_constructions_kernel,
                                                        SkeletonK ssk = CGAL::Exact_predicates_inexact_constructions_kernel)
```

returns a container with all the inner offset polygons at distance $offset$ of the 2D polygon with holes whose outer boundary is *outer_boundary* and its holes are given by *[holes_begin,holes_end]*. A temporary straight skeleton is constructed in the interior of the input polygon to obtain the offsets. The construction of this skeleton is the most expensive operation, therefore, to construct offsets at more than one single distance, use the separate functions *create_interior_straight_skeleton_2* and *create_offset_polygons_2* instead.

```
template<class FT, class Polygon, class OffsettingK, class SkeletonK>
std::vector< boost::shared_ptr<Polygon> >
        create_interior_skeleton_and_offset_polygons_2( FT offset,
                                                        Polygon poly,
                                                        OffsettingK ofk = CGAL::Exact_predicates_inexact_constructions_kernel,
                                                        SkeletonK ssk = CGAL::Exact_predicates_inexact_constructions_kernel)
```

returns a container with all the inner offset polygons at distance $offset$ of the 2D polygon *poly*. A temporary straight skeleton is constructed in the interior of the input polygon to obtain the offsets. The construction of this skeleton is the most expensive operation, therefore, to construct offsets at more than one single distance, use the separate functions *create_interior_straight_skeleton_2* and *create_offset_polygons_2* instead.

#include <CGAL/create_offset_polygons_from_polygon_with_holes_2.h>

template<class FT, class OffsettingK, class SkeletonK>
std::vector< boost::shared_ptr< Polygon_2<OffsettingK> > >

create_interior_skeleton_and_offset_polygons_2(FT offset, CGAL::Polygon_with_holes< OffsettingK> poly_with_holes, OffsettingK ofk = CGAL::Exact_predicates_inexact_constructions_kernel, SkeletonK ssk = CGAL::Exact_predicates_inexact_constructions_kernel)

returns a container with all the inner offset polygons at distance $offset$ of the 2D polygon with holes $poly_with_holes$. A temporary straight skeleton is constructed in the interior of the input polygon to obtain the offsets. The construction of this skeleton is the most expensive operation, therefore, to construct offsets at more than one single distance, use the separate functions *create_interior_straight_skeleton_2* and *create_offset_polygons_2* instead.

template<class FT, class OffsettingK, class SkeletonK>
std::vector< boost::shared_ptr< Polygon_with_holes_2<OffsettingK> > >

create_interior_skeleton_and_offset_polygons_with_holes_2(FT offset, CGAL::Polygon_2< OffsettingK> poly, OffsettingK ofk = CGAL::Exact_predicates_inexact_constructions_kernel, SkeletonK ssk = CGAL::Exact_predicates_inexact_constructions_kernel)

returns a container with all the inner offset polygons *with holes* at distance $offset$ of the 2D polygon $poly$. This is equivalent to *arrange_offset_polygons_2(create_interior_skeleton_and_offset_polygons_2(offset,poly,okk,ssk))*.

template<class FT, class OffsettingK, class SkeletonK>
std::vector< boost::shared_ptr< Polygon_with_holes_2<OffsettingK> > >

create_interior_skeleton_and_offset_polygons_with_holes_2(FT offset, CGAL::Polygon_with_holes_2<OffsettingK> poly_with_holes, OffsettingK ofk = CGAL::Exact_predicates_inexact_constructions_kernel, SkeletonK ssk = CGAL::Exact_predicates_inexact_constructions_kernel)

returns a container with all the inner offset polygons *with holes* at distance $offset$ of the 2D polygon with holes $poly_with_holes$. This is equivalent to *arrange_offset_polygons_2(create_interior_skeleton_and_offset_polygons_2(offset,poly_with_holes,okk,ssk))*.

1. *OffsettingK* is the CGAL kernel used to instantiate *Polygon_offset_builder_traits_2<K>* for constructing the offset polygons.

2. *SkeletonK* is the CGAL kernel used to instantiate *Straight_skeleton_builder_traits_2<K>* for constructing the straight skeleton. If *SkeletonK != OffsettingK* the constructed straight skeleton is converted to *Straight_skeleton_2<OffsettingK>*.

3. *FT* is any number type implicitly convertible to *OffsettingK::FT*.

4. *Straight_skeleton* is *Straight_skeleton_2<K2>*. If $K! = K2$ the straight skeleton is converted to *Straight_skeleton_2<K>*.

5. *HoleIterator::value_type* and *Polygon* are *CGAL::Polygon_2<OffsettingK>* or a standard container of *OffsettingK::Point_2* elements

See Also

create_exterior_straight_skeleton_2
Straight_skeleton_builder_2<Gt,Ss>

CGAL::create_exterior_skeleton_and_offset_polygons_2

Definition

The function *create_exterior_skeleton_and_offset_polygons_2* creates outer offset polygons directly from an input polygon via a temporarily created straight skeleton.

#include <CGAL/create_offset_polygons_2.h>

```
template<class FT, class Polygon, class OffsettingK, class SkeletonK>
std::vector< boost::shared_ptr<Polygon> >

        create_exterior_skeleton_and_offset_polygons_2( FT offset,
                                                        Polygon poly,
                                                        OffsettingK ofk = Exact_predicates_inexact_constructions_kernel,
                                                        SkeletonK ssk = Exact_predicates_inexact_constructions_kernel)
```

returns a container with all the outer offset polygons at distance $offset$ of the 2D polygon *poly*. A temporary straight skeleton is constructed in the *limited exterior* of the input polygon to obtain the offsets. The construction of this skeleton is the most expensive operation, therefore, to construct offsets at more than one single distance, use the separate functions *create_exterior_straight_skeleton_2* and *create_offset_polygons_2* instead. The exterior skeleton is limited by an outer rectangular frame placed at a margin sufficiently large to allow the offset polygons to be constructed.

#include <CGAL/create_offset_polygons_from_polygon_with_holes_2.h>

```
template<class FT, class OffsettingK, class SkeletonK>
std::vector< boost::shared_ptr< Polygon_with_holes_2<OffsettingK> > >

        create_exterior_skeleton_and_offset_polygons_with_holes_2( FT offset,
                                                                   CGAL::Polygon_2< OffsettingK> poly_with_holes,
                                                                   OffsettingK ofk = Exact_predicates_inexact_constructions_kernel,
                                                                   SkeletonK ssk = Exact_predicates_inexact_constructions_kernel)
```

returns a container with all the outer offset polygons *with holes* at distance $offset$ of the 2D polygon *poly*. This is equivalent to *arrange_offset_polygons_2(create_exterior_skeleton_and_offset_polygons_2(offset,poly,okk,ssk))*.

Requirements

1. *OffsettingK* is the CGAL kernel used to instantiate *Polygon_offset_builder_traits_2<K>* for constructing the offset polygons.

2. *SkeletonK* is the CGAL kernel used to instantiate *Straight_skeleton_builder_traits_2<K>* for constructing the straight skeleton. If *SkeletonK != OffsettingK* the constructed straight skeleton is converted to *Straight_skeleton_2<OffsettingK>*.

3. *FT* is any number type implicitly convertible to *OffsettingK::FT*.

4. *Straight_skeleton* is *Straight_skeleton_2<K2>*. If $K! = K2$ the straight skeleton is converted to *Straight_skeleton_2<K>*.

See Also

create_exterior_straight_skeleton_2
Straight_skeleton_builder_2<Gt,Ss>

Class

CGAL::Polygon_offset_builder_traits_2<Kernel>

Definition

The class *Polygon_offset_builder_traits_2<Kernel>* provides a model for the *PolygonOffsetBuilderTraits_2* concept which is the traits class required by the *Polygon_offset_builder_2* algorithm class. The class *Polygon_offset_builder_traits_2<Kernel>* has one template argument: a 2D CGAL Kernel. This parameter must be a model for the *Kernel* concept, such as the *Exact_predicates_inexact_constructions_kernel*, which is the recommended one. It is unspecified which subset of the kernel is used in this traits class (and by extension in the builder class). This is to avoid restricting the choices in the implementation.

#include <CGAL/Polygon_offset_builder_traits_2.h>

Is Model for the Concepts

PolygonOffsetBuilderTraits_2
DefaultConstructible
CopyConstructible

See Also

CGAL::Polygon_offset_builder_2<Ss,Gt,Polygon_2>

CGAL::Polygon_offset_builder_2<Ss,Gt,Container>

Class

Definition

The class *Polygon_offset_builder_2<Ss,Gt,Container>* encapsulates the construction of inward offset contours of a 2D simple polygon with holes. The construction is based on the straight skeleton of the interior of the polygon. Its first template parameter, *Ss*, must be a model of the *StraightSkeleton_2* concept, its second template parameter, *Gt*, must be a model of the *StraightSkeletonBuilderTraits_2* concept, and its third template parameter must be a model of the *VertexContainer_2* concept.

#include <CGAL/Polygon_offset_builder_2.h>

Types

Polygon_offset_builder_2<Ss,Gt,Container>:: Ss

The straight skeleton (first template parameter)

Polygon_offset_builder_2<Ss,Gt,Container>:: Gt

The geometric traits (second template parameter)

Polygon_offset_builder_2<Ss,Gt,Container>:: Container

The container of 2D vertices that represents each offset contour generated by the algorithm (third template parameter)

Polygon_offset_builder_2<Ss,Gt,Container>:: FT

The *SqrtFieldNumberType* used to specify the desired offset distance, provided by the geometric traits *Gt*.

Creation

Polygon_offset_builder_2<Ss,Gt,Container> b(Ss ss);

Constructs the builder class using the given Straight Skeleton instance.

Methods

template<class OutputIterator>

OutputIterator *b.construct_offset_contours(FT t, OutputIterator out)*

Given the straight skeleton passed in the constructor which corresponds to the interior of a non-degenerate strictly-simple polygon with holes P, returns *all* the offset contours of P at the Euclidean distance t.

Such offset contours are polygons in the interior of P. Provided exact constructions are used, these polygons are simple.

For any offset distance t there are 0, 1 or more offset contours.

For each resulting offset contour, a default constructed instance of *Container* type (which must be a model of the *VertexContainer_2* concept), is dynamically allocated and each offset vertex is added to it.

A *boost::shared_ptr* holding onto the dynamically allocated container is inserted into the output sequence via the OutputIterator *out*.

OutputIterator must be a model of the *OutputIterator* category whose *value_type* is a *boost::shared_ptr* holding the dynamically allocated instances of type Container.

The method returns an OutputIterator past-the-end of the resulting sequence, which contains each offset contour generated.

You can call *construct_offset_contours* with different offset distances (there is no need to construct the builder again). If you call it with an offset distance so large that there are no offset contours at that distance, no contour is inserted into the output sequence and the returned iterator will be equal to *out*.

For any given input polygon, its offset polygons at a certain distance are composed of several contours. This method returns all such contours in an unspecified order and with no parental relationship between them (that is why it is called *construct_offset_contours* and not *construct_offset_polygons*).

Those offset contours in the resulting sequence which are oriented counter-clockwise are outer contours and those oriented clockwise are holes. It is up to the user to match each hole to its parent in order to reconstruct the parent-hole relationship of the conceptual output. It is sufficient to test each hole against each parent as there won't be a hole inside a hole, a parent inside any other contour, or a hole inside more than one parent. The recommended insideness test is to perform a regularized Boolean set operation. *do_intersect* from the *Boolean Set Operations* package will work fine in our case since it is guaranteed that no hole would cross halfway outside any parent (in the presence of such cases, subtracting the parent from the hole works better as it correctly rejects halfway holes).

If there are degenerate, or nearly degenerate vertices; that is, vertices whose internal or external angle approaches *0*, numerical overflow may prevent some of the polygon contours to be constructed. If that happens, the failed contour just won't be added into the resulting sequence.

See Also

VertexContainer_2
PolygonOffsetBuilderTraits_2
CGAL::Polygon_offset_builder_traits_2<Kernel>

Function

CGAL::compute_outer_frame_margin

Definition

The function *compute_outer_frame_margin* computes the separation required between a polygon and the outer frame used to obtain an exterior skeleton suitable for the computation of outer offset polygons at a given distance.

#include <CGAL/compute_outer_frame_margin.h>

template <class InputIterator, class Traits>
boost::optional< typename Traits::FT >

compute_outer_frame_margin(InputIterator first,
InputIterator beyond,
typename Traits::FT offset,
Traits traits = Default_traits)

Given a non-degenerate strictly-simple 2D polygon whose vertices are passed in the range [*first*,*beyond*), calculates the largest euclidean distance *d* between each input vertex and its corresponding offset vertex at a distance *offset*.
If such a distance can be approximately computed, returns an *optional<FT>* with the value *d + (offset * 1.05)*. If the distance cannot be computed, not even approximately, due to overflow for instance, returns an empty *optional<FT>* (an *absent result*).
This result is the required separation between the input polygon and the rectangular frame used to construct an exterior offset contour at distance *offset* (which is done by placing the polygon as a hole of that frame).
Such a separation must be computed in this way because if the frame is too close to the polygon, the inward offset contour from the frame could collide with the outward offset contour of the polygon, resulting in a merged contour offset instead of two contour offsets, one of them corresponding to the frame.
Simply using *2*offset* as the separation is incorrect since *offset* is the distance between an offset line and its original, not between an offset vertex and its original. The later, which is calculated by this function and needed to place the frame sufficiently away from the polygon, can be thousands of times larger than *offset*.
If the result is *absent*, any attempt to construct an exterior offset polygon at distance *offset* will fail. This will occur whenever the polygon has a vertex with an internal angle approaching *0* (because the offset vertex of a vertex whose internal angle equals 0 is at *infinity*).

Precondition: offset > 0.
Precondition: The range [*first*,*beyond*) contains the vertices of a non-degenerate strictly-simple 2D polygon.

The default traits class *Default_traits* is an instance of the class *Polygon_offset_builder_traits_2<Kernel>* parameterized on the kernel in which the type *InputIterator::value_type* is defined.

Requirements

1. *InputIterator::value_type* is equivalent to *Traits::Point_2*.
2. *Traits* must be a model for *PolygonOffsetBuilderTraits_2*

See Also

PolygonOffsetBuilderTraits_2
CGAL::Polygon_offset_builder_traits_2<Kernel>

Chapter 24

2D Minkowski Sums

Ron Wein

Contents

24.1 Introduction

Given two sets $A, B \in \mathbb{R}^d$, their *Minkowski sum*, denoted by $A \oplus B$, is the set $\{a + b \mid a \in A, b \in B\}$. Minkowski sums are used in many applications, such as motion planning and computer-aided design and manufacturing. This package contains functions for computing planar Minkowski sums of two polygons (namely A and B are two closed polygons in $\mathbb{R}^2$), and for a polygon and a disc (an operation also known as *offsetting* or *dilating* a polygon).

24.2 Computing the Minkowski Sum of Two Polygons

Computing the Minkowski sum of two convex polygons P and Q with m and n vertices respectively is very easy, as $P \oplus Q$ is a convex polygon bounded by copies of the $m + n$ edges, and these edges are sorted by the angle they form with the x-axis. As the two input polygons are convex, their edges are already sorted by the angle they form with the x-axis. The Minkowski sum can therefore be computed in $O(m + n)$ time, by starting from two bottommost vertices in P and in Q and performing "merge sort" on the edges.

If the polygons are not convex, it is possible to use one of the following approaches:

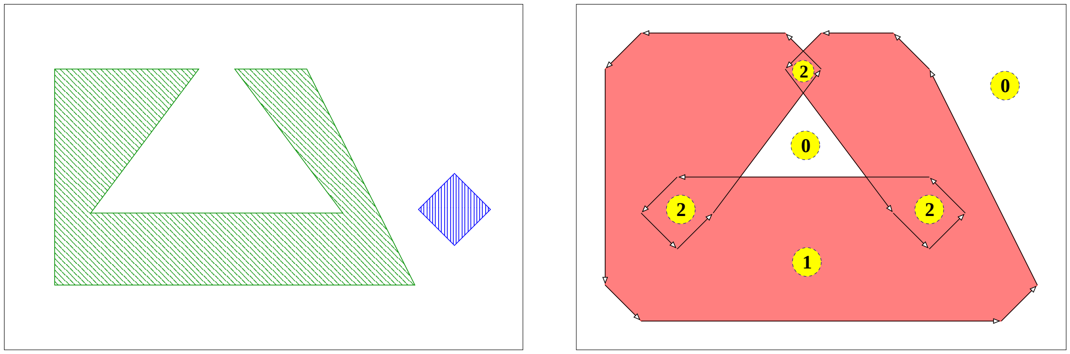


Figure 24.1: Computing the convolution of a convex polygon and a non-convex polygon (left). The convolution consists of a single self-intersecting cycle, drawn as a sequence of arrows (right). The winding number associated with each face of the arrangement induced by the segments forming the cycle appears in dashed circles. The Minkowski sum of the two polygons is shaded.

Decomposition: We decompose P and Q into convex sub-polygons, namely we obtain two sets of convex polygons $P_1, \ldots, P_k$ and $Q_1, \ldots, Q_\ell$ such that $\bigcup_{i=1}^{k} P_i = P$ and $\bigcup_{i=j}^{\ell} Q_j = Q$. We then calculate the pairwise sums $S_{ij} = P_i \oplus Q_j$ using the simple procedure described above, and compute the union $P \oplus Q = \bigcup_{ij} S_{ij}$.

This approach relies on a decomposition strategy that computes the convex decomposition of the input polygons and its performance depends on the quality of the decomposition.

Convolution: Let us denote the vertices of the input polygons by $P = (p_0, \ldots, p_{m-1})$ and $Q = (q_0, \ldots, q_{n-1})$. We assume that both P and Q have positive orientations (i.e. their boundaries wind in a counterclockwise order around their interiors) and compute the convolution of the two polygon boundaries. The *convolution* of these two polygons [GRS83], denoted $P * Q$, is a collection of line segments of the form $[p_i + q_j, p_{i+1} + q_j]$,[1] where the vector $\mathbf{p_i p_{i+1}}$ lies between $\mathbf{q_{j-1} q_j}$ and $\mathbf{q_j q_{j+1}}$,[2] and — symmetrically — of segments of the form $[p_i + q_j, p_i + q_{j+1}]$, where the vector $\mathbf{q_j q_{j+1}}$ lies between $\mathbf{p_{i-1} p_i}$ and $\mathbf{p_i p_{i+1}}$.

The segments of the convolution form a number of closed (not necessarily simple) polygonal curves called *convolution cycles*. The Minkowski sum $P \oplus Q$ is the set of points having a non-zero winding number with respect to the cycles of $P * Q$.[3] See Figure 24.1 for an illustration.

The number of segments in the convolution of two polygons is usually smaller than the number of segments that constitute the boundaries of the sub-sums S_{ij} when using the decomposition approach. As both approaches construct the arrangement of these segments and extract the sum from this arrangement, computing Minkowski sum using the convolution approach usually generates a smaller intermediate arrangement, hence it is faster and consumes less space.

24.2.1 Computing Minkowski Sum using Convolutions

The function *minkowski_sum_2 (P, Q)* accepts two simple polygons P and Q, represented using the *Polygon_2<Kernel,Container>* class-template and uses the convolution method in order to compute and return their Minkowski sum $S = P \oplus Q$.

As the input polygons may not be convex, their Minkowski sum may not be simply connected and contain polygonal holes; see for example Figure 24.1. S is therefore an instance of the *Polygon_with_holes_2< Kernel,Container>* class-template, defined in the Boolean Set-Operations package: The outer boundary of S

[1] Throughout this chapter, we increment or decrement an index of a vertex modulo the size of the polygon.

[2] We say that a vector $\mathbf{v}$ lies between two vectors $\mathbf{u}$ and $\mathbf{w}$ if we reach $\mathbf{v}$ strictly before reaching $\mathbf{w}$ if we move all three vectors to the origin and rotate $\mathbf{u}$ counterclockwise. Note that this also covers the case where $\mathbf{u}$ has the same direction as $\mathbf{v}$.

[3] Informally speaking, the winding number of a point $p \in \mathbb{R}^2$ with respect to some planar curve γ is an integer number counting how many times does γ wind in a counterclockwise direction around p.

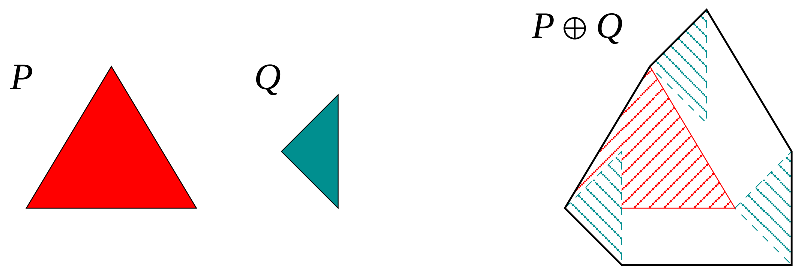


Figure 24.2: Computing the Minkowski sum of two triangles, as done in the example program *Minkowski_sum_2/sum_triangles.cpp*.

is a polygon that can be accessed using *S.outer_boundary()*, and its polygonal holes are given by the range *[S.holes_begin(), S.holes_end())* (where *S* contains *S.number_of_holes()* holes in its interior).

The following example program constructs the Minkowski sum of two triangles, as depicted in Figure 24.2. The result in this case is a convex hexagon. This program, as other example programs in this chapter, includes the auxiliary header file *ms_rational_nt.h* which defines *Number_type* as either *Gmpq* or *Quotient<MP_Float>*, depending on whether the GMP library is installed or not. The file *print_util.h* includes auxiliary functions for printing polygons.

```
#include "ms_rational_nt.h"
#include <CGAL/Cartesian.h>
#include <CGAL/minkowski_sum_2.h>
#include <iostream>

#include "print_utils.h"

typedef CGAL::Cartesian<Number_type>                Kernel;
typedef Kernel::Point_2                             Point_2;
typedef CGAL::Polygon_2<Kernel>                     Polygon_2;
typedef CGAL::Polygon_with_holes_2<Kernel>          Polygon_with_holes_2;

int main ()
{
  // Construct the first polygon (a triangle).
  Polygon_2   P;

  P.push_back (Point_2 (0, 0));
  P.push_back (Point_2 (6, 0));
  P.push_back (Point_2 (3, 5));

  // Construct the second polygon (a triangle).
  Polygon_2   Q;

  Q.push_back (Point_2 (0, 0));
  Q.push_back (Point_2 (2, -2));
  Q.push_back (Point_2 (2, 2));

  // Compute the Minkowski sum.
  Polygon_with_holes_2  sum = minkowski_sum_2 (P, Q);
```

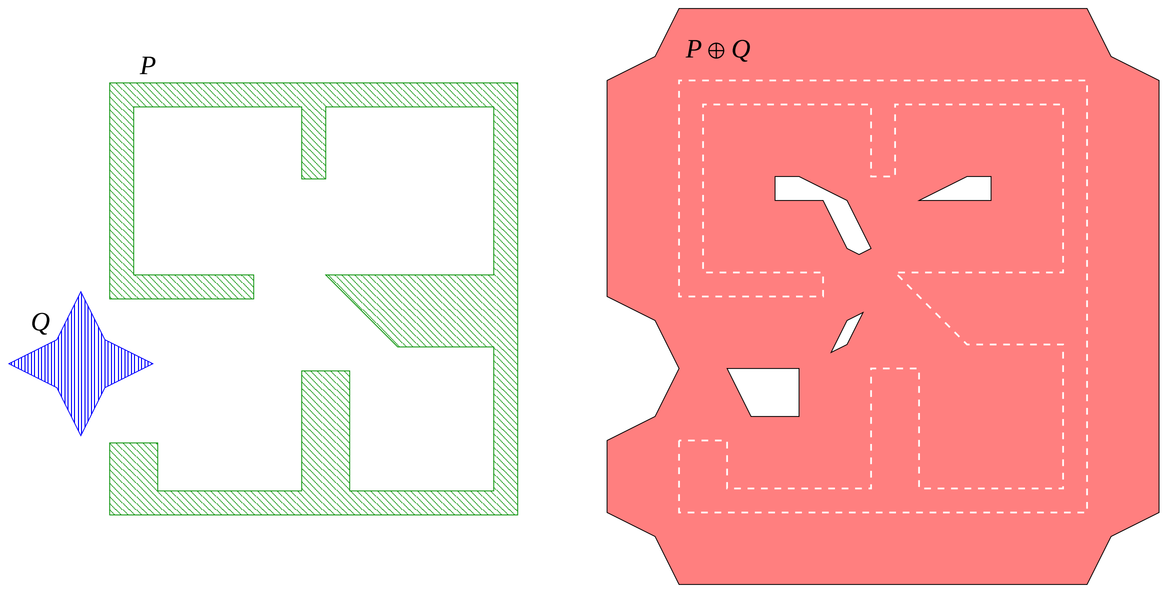


Figure 24.3: Computing the Minkowski sum of two non-convex polygons P and Q, as done in the example programs *Minkowski_sum_2/sum_with_holes.cpp* and *Minkowski_sum_2/sum_by_decomposition.cpp*.

```
  CGAL_assertion (sum.number_of_holes() == 0);

  std::cout << "P = "; print_polygon (P);
  std::cout << "Q = "; print_polygon (Q);
  std::cout << "P (+) Q = "; print_polygon (sum.outer_boundary());

  return (0);
}
```

File: examples/Minkowski_sum_2/sum_triangles.cpp

In the following program we compute the Minkowski sum of two polygons that are read from an input file. In this case, the sum is not simple and contains four holes, as illustrated in Figure 24.3. Note that this example uses the predefined CGAL kernel with exact constructions. In general, instantiating polygons with this kernel yields the fastest running times for Minkowski-sum computations.

```
#include <CGAL/Exact_predicates_exact_constructions_kernel.h>
#include <CGAL/minkowski_sum_2.h>
#include <iostream>
#include <fstream>

#include "print_utils.h"

struct Kernel : public CGAL::Exact_predicates_exact_constructions_kernel
   {};

typedef Kernel::Point_2                                    Point_2;
typedef CGAL::Polygon_2<Kernel>                            Polygon_2;
```

```
typedef CGAL::Polygon_with_holes_2<Kernel>
    Polygon_with_holes_2;

int main ()
{
  // Open the input file.
  std::ifstream    in_file ("rooms_star.dat");

  if (! in_file.is_open())
  {
    std::cerr << "Failed to open the input file." << std::endl;
    return (1);
  }

  // Read the two polygons from the file and compute their Minkowski sum.
  Polygon_2    P, Q;

  in_file >> P >> Q;
  in_file.close();

  // Compute and print the Minkowski sum.
  Polygon_with_holes_2  sum = minkowski_sum_2 (P, Q);

  std::cout << "P (+) Q = "; print_polygon_with_holes (sum);

  return (0);
}
```

File: examples/Minkowski_sum_2/sum_with_holes.cpp

24.2.2 Decomposition Strategies

In order to compute Minkowski sums using the decomposition method, it is possible to call the function *minkowski_sum_2 (P, Q, decomp)*, where *decomp* is an instance of a class that models the concept *PolygonConvexDecomposition_2*, namely it should provide a method named *decompose()* that receives a planar polygon and returns a range of convex polygons that represents its convex decomposition.

The Minkowski-sum package includes four models of the concept *PolygonConvexDecomposition_2*. The first three are classes that wrap the decomposition functions included in the Planar Polygon Partitioning package, while the fourth is an implementation of a decomposition algorithm introduced in [AFH02]. The convex decompositions that it creates usually yield efficient running times for Minkowski sum computations:

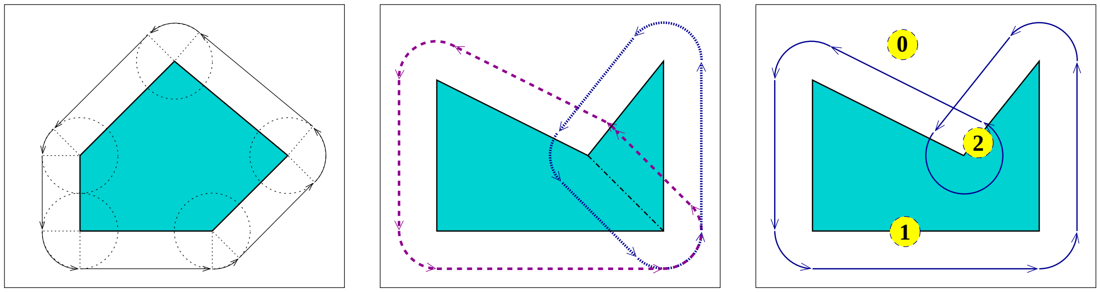
- The class *Optimal_convex_decomposition<Kernel>* uses the dynamic-programming algorithm of Greene [Gre83] for computing an optimal decomposition of a polygon into a minimal number of convex sub-polygons. The main drawback of this strategy is that it runs in $O(n^4)$ time and $O(n^3)$ in the worst case,where n is the number of vertices in the input polygon.
- The class *Hertel_Mehlhorn_convex_decomposition<Kernel>* implements the approximation algorithm suggested by Hertel and Mehlhorn [HM83], which triangulates the input polygon and proceeds by throwing away unnecessary triangulation edges. This algorithm requires $O(n)$ time and space and guarantees that the number of sub-polygons it generates is not more than four times the optimum.

- The class *Greene_convex_decomposition<Kernel>* is an implementation of Greene's approximation algorithm [Gre83], which computes a convex decomposition of the polygon based on its partitioning into y-monotone polygons. This algorithm runs in $O(n \log n)$ time and $O(n)$ space, and has the same approximation guarantee as Hertel and Mehlhorn's algorithm.

- The class *Small_side_angle_bisector_convex_decomposition<Kernel>* uses a heuristic improvement to the angle-bisector decomposition method suggested by Chazelle and Dobkin [CD85], which runs in $O(n^2)$ time. It starts by examining each pair of reflex vertices in the input polygon such that the entire interior of the diagonal connecting these vertices is contained in the polygon. Out of all available pairs, the vertices p_i and p_j are selected such that the number of reflex vertices from p_i to p_j (or from p_j to p_i) is minimal. The polygon is split by the diagonal $p_i p_j$ and we continue recursively on both resulting sub-polygons. In case it is not possible to eliminate two reflex vertices at once any more, each reflex vertex is eliminated by a diagonal that is closest to the angle bisector emanating from this vertex.

The following example demonstrates the computation of the Minkowski sum of the same input polygons as used in *Minkowski_sum_2/sum_with_holes.cpp* (as depicted in Figure 24.3), using the small-side angle-bisector decomposition strategy:

```
#include <CGAL/Exact_predicates_exact_constructions_kernel.h>
#include <CGAL/minkowski_sum_2.h>
#include <CGAL/Small_side_angle_bisector_decomposition_2.h>
#include <iostream>
#include <fstream>

#include "print_utils.h"

struct Kernel : public CGAL::Exact_predicates_exact_constructions_kernel
   {};

typedef Kernel::Point_2                                  Point_2;
typedef CGAL::Polygon_2<Kernel>                          Polygon_2;
typedef CGAL::Polygon_with_holes_2<Kernel>
   Polygon_with_holes_2;

int main ()
{
  // Open the input file.
  std::ifstream    in_file ("rooms_star.dat");

  if (! in_file.is_open())
  {
    std::cerr << "Failed to open the input file." << std::endl;
    return (1);
  }

  // Read the two polygons from the file and compute their Minkowski sum.
  Polygon_2   P, Q;

  in_file >> P >> Q;
  in_file.close();

  // Compute the Minkowski sum using the decomposition approach.
  CGAL::Small_side_angle_bisector_decomposition_2<Kernel>  ssab_decomp;
```



(a) (b) (c)

Figure 24.4: (a) Offsetting a convex polygon. (b) Computing the offset of a non-convex polygon by decomposing it to convex sub-polygons. (c) Offsetting a non-convex polygon by computing its convolution with a disc. The convolution cycle induces an arrangement with three faces, whose winding numbers are shown enclosed in dashed circles.

```
  Polygon_with_holes_2  sum = minkowski_sum_2 (P, Q, ssab_decomp);

  std::cout << "P (+) Q = "; print_polygon_with_holes (sum);

  return (0);
}
```

File: `examples/Minkowski_sum_2/sum_by_decomposition.cpp`

24.3 Offsetting a Polygon

The operation of computing the Minkowski sum $P \oplus B_r$ of a polygon P with b_r, a disc of radius r centered at the origin, is widely known as *offsetting* the polygon P by a radius r.

Let $P = (p_0, \ldots, p_{n-1})$ be the polygon vertices, ordered in a counterclockwise orientation around its interior. If P is a convex polygon the offset is easily computed by shifting each polygon edge by r away from the polygon, namely to the right side of the edge. As a result we obtain a collection of n disconnected *offset edges*. Each pair of adjacent offset edges, induced by $p_{i-1}p_i$ and p_ip_{i+1}, are connected by a circular arc of radius r, whose supporting circle is centered at p_i. The angle that defines such a circular arc equals $180^\circ - \angle(p_{i-1}, p_i, p_{i+1})$; see Figure 24.4(a) for an illustration. The running time of this simple process is of course linear with respect to the size of the polygon.

If P is not convex, its offset can be obtained by decomposing it to convex sub-polygons $P_1, \ldots P_m$ such that $\bigcup_{i=1}^{m} P_i = P$, computing the offset of each sub-polygon and finally calculating the union of these sub-offsets (see Figure 24.4(b)). However, as was the case with the Minkowski sum of a pair of polygons, here it is also more efficient to compute the *convolution cycle* of the polygon with the disc B_r,[4] which can be constructed by applying the process described in the previous paragraph. The only difference is that a circular arc induced by a reflex vertex p_i is defined by an angle $540^\circ - \angle(p_{i-1}, p_i, p_{i+1})$; see Figure 24.4(c) for an illustration. We finally compute the winding numbers of the faces of the arrangement induced by the convolution cycle to obtain the offset polygon.

[4]As the disc is convex, it is guaranteed that the convolution curve is comprised of a single cycle.

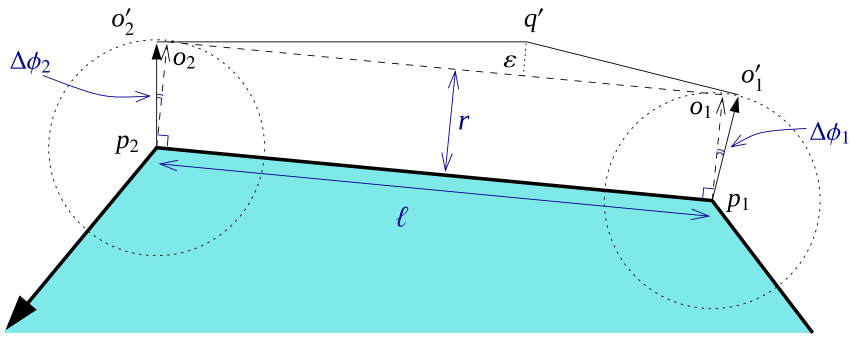


Figure 24.5: Approximating the offset edge $o_1 o_2$ induced by the polygon edge $p_1 p_2$ by two line segments $o'_1 q'$ and $q' o'_2$.

24.3.1 Approximating the Offset with a Guaranteed Error-Bound

Let us assume that we are given a counterclockwise-oriented polygon $P = (p_0, \ldots, p_{n-1})$, whose vertices all have rational coordinates (i.e., for each vertex $p_i = (x_i, y_i)$ we have $x_i, y_i \in \mathbb{Q}$, and we wish to compute its Minkowski sum with a disc of radius r, where r is also a rational number. The boundary of this sum is comprised of line segments and circular arcs, where:

- Each circular arc is supported by a circle of radius r centered at one of the polygon vertices. The equation of this circle is $(x - x_i)^2 + (y - y_i)^2 = r^2$, and has only rational coefficients.
- Each line segment is supported by a line parallel to one of the polygon edges $p_i p_{i+1}$, which lies at distance r from this edge. If we denote the supporting line of $p_i p_{i+1}$ by $ax + by + c = 0$, where $a, b, c \in \mathbb{Q}$, then the offset edge is supported by the line $ax + by + (c + \ell \cdot r) = 0$, where ℓ is the length of the edge $p_i p_{i+1}$ and is in general *not* a rational number. The line segments that comprise the offset boundaries therefore cannot be represented as segments of lines with rational coefficients. We mention that the locus of all points that lie at distance r from the line $ax + by + c = 0$ is given by:

  $$\frac{(ax + by + c)^2}{a^2 + b^2} = r^2 \, .$$

  Thus, the linear offset edges are segments of curves of an algebraic curve of degree 2 (a conic curve) with rational coefficients. This curve is actually a pair of the parallel lines $ax + by + (c \pm \ell \cdot r) = 0$. In Section 24.3.2 we use this representation to construct the offset polygon in an exact manner using the traits class for conic arcs.

The class-template *Gps_circle_segment_traits_2<Kernel>*, included in the Boolean Set-Operations package is used for representing general polygons whose edges are circular arcs or line segments, and for applying set operations (e.g. intersection, union, etc.) on such general polygon. It should be instantiated with a geometric kernel that employs exact rational arithmetic, such that the curves that comprise the polygon edges should be arcs of circles with rational coefficients or segments of lines with rational coefficients. As in our case the line segments do not satisfy this requirement, we apply a simple approximation scheme, such that each irrational line segment is approximated by two rational segments:

1. Consider the example depicted in Figure 24.5, where the exact offset edge $o_1 o_2$ is obtained by shifting the polygon edge $p_1 p_2$ by a vector whose length is r that form an angle ϕ with the x-axis. We select two points o'_1 and o'_2 with rational coordinates on the two circles centered at p_1 and p_2, respectively. These points are selected such that if we denote the angle that the vector $\mathbf{p_j o_j}$ forms with the x-axis by ϕ'_j (for $j = 1, 2$), we have $\phi'_1 < \phi < \phi'_2$.

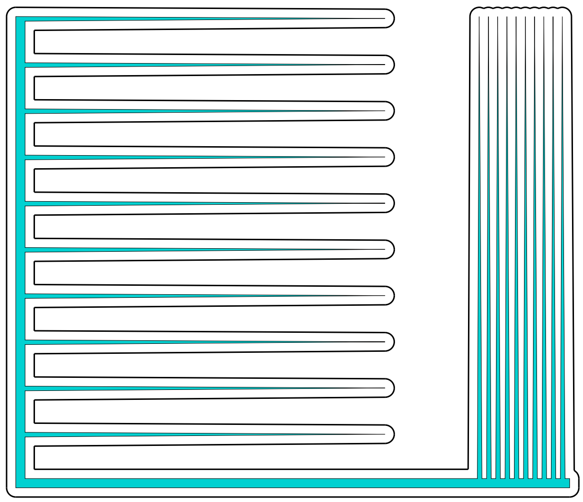


Figure 24.6: The offset computation performed by the example programs *Minkowski_sum_2/approx_offset.cpp* and *Minkowski_sum_2/exact_offset.cpp*. The input polygon is shaded and the boundary of its offset is drawn in a thick black line.

2. We construct two tangents to the two circles at o'_1 and o'_2, respectively. The tangent lines have rational coefficients.

3. We compute the intersection point of the two tangents, denoted q'. The two line segments $o'_1 q'$ and $q' o'_2$ approximate the original offset edge $o_1 o_2$.

The function *approximated_offset_2 (P, r, eps)* accepts a polygon P, an offset radius r and $\varepsilon > 0$. It constructs an approximation for the offset of P by the radius r using the procedure explained above. Furthermore, it is guaranteed that the approximation error, namely the distance of the point q' from $o_1 o_2$ is bounded by ε. Using this function, we are capable of working with the *Gps_circle_segment_traits_2<Kernel>* class, which considerably speeds up the (approximate) construction of the offset polygon and the application of set operations on such polygons. The function returns an object of the nested type *Gps_circle_segment_traits_2<Kernel> ::Polygon_with_holes_2* representing the approximated offset polygon (recall that if P is not convex, its offset may not be simple and may contain holes, whose boundary is also comprised of line segments and circular arcs).

The following example demonstrates the construction of an approximated offset of a non-convex polygon, as depicted in Figure 24.6. Note that we use a geometric kernel parameterized with a filtered rational number-type. Using filtering considerably speeds up the construction of the offset.

```
#include "ms_rational_nt.h"
#include <CGAL/Lazy_exact_nt.h>
#include <CGAL/Cartesian.h>
#include <CGAL/approximated_offset_2.h>
#include <CGAL/offset_polygon_2.h>
#include <CGAL/Timer.h>
#include <iostream>

typedef CGAL::Lazy_exact_nt<Number_type>           Lazy_exact_nt;

struct Kernel : public CGAL::Cartesian<Lazy_exact_nt> {};
```

```
typedef CGAL::Polygon_2<Kernel>                          Polygon_2;

typedef CGAL::Gps_circle_segment_traits_2<Kernel>  Gps_traits_2;
typedef Gps_traits_2::Polygon_2                          Offset_polygon_2;
typedef Gps_traits_2::Polygon_with_holes_2
    Offset_polygon_with_holes_2;

int main ()
{
  // Open the input file.
  std::ifstream    in_file ("spiked.dat");

  if (! in_file.is_open())
  {
    std::cerr << "Failed to open the input file." << std::endl;
    return (1);
  }

  // Read the input polygon.
  Polygon_2        P;

  in_file >> P;
  in_file.close();

  std::cout << "Read an input polygon with "
            << P.size() << " vertices." << std::endl;

  // Approximate the offset polygon.
  const Number_type             radius = 5;
  const double                  err_bound = 0.00001;
  Offset_polygon_with_holes_2  offset;
  CGAL::Timer                   timer;

  timer.start();
  offset = approximated_offset_2 (P, radius, err_bound);
  timer.stop();

  std::cout << "The offset polygon has "
            << offset.outer_boundary().size() << " vertices, "
            << offset.number_of_holes() << " holes." << std::endl;
  std::cout << "Offset computation took "
            << timer.time() << " seconds." << std::endl;
  return (0);
}
```

File: examples/Minkowski_sum_2/approx_offset.cpp

24.3.2 Computing the Exact Offset

As we previously mentioned, it is possible to represent offset polygons in an exact manner, if we treat their edges as arcs of conic curves with rational coefficients. The function *offset_polygon_2 (P, r, traits)* computes the offset of the polygon P by a rational radius r in an exact manner. The *traits* parameter should be an instance of an arrangement-traits class that is capable of handling conic arcs in an exact manner; using the *Arr_conic_traits_2* class with the number types provided by the CORE library is the preferred option. The function returns an object of the nested type *Gps_traits_2<ArrConicTraits>::Polygons_with_holes_2* (see the documentation of the Boolean Set-Operations package for more details on the traits-class adapter *Gps_traits_2*), which represented the exact offset polygon.

The following example demonstrates the construction of the offset of the same polygon that serves as an input for the example program *Minkowski_sum_2/approx_offset.cpp*, presented in the previous subsection (see also Figure 24.6). Note that the resulting polygon is smaller than the one generated by the approximated-offset function (recall that each irrational line segment in this case is approximated by two rational line segments), but the offset computation is considerably slower:

```
#include <CGAL/basic.h>

#ifndef CGAL_USE_CORE
#include <iostream>
int main ()
{
  std::cout << "Sorry, this example needs CORE ..." << std::endl;
  return (0);
}
#else

#include <CGAL/Cartesian.h>
#include <CGAL/CORE_algebraic_number_traits.h>
#include <CGAL/Arr_conic_traits_2.h>
#include <CGAL/offset_polygon_2.h>
#include <CGAL/Timer.h>
#include <iostream>

typedef CGAL::CORE_algebraic_number_traits     Nt_traits;
typedef Nt_traits::Rational                    Rational;
typedef Nt_traits::Algebraic                   Algebraic;

struct Rat_kernel : public CGAL::Cartesian<Rational> {};
struct Alg_kernel : public CGAL::Cartesian<Algebraic> {};
struct Conic_traits_2 : public CGAL::Arr_conic_traits_2<Rat_kernel,
                                                        Alg_kernel,
                                                        Nt_traits> {};

typedef CGAL::Polygon_2<Rat_kernel>            Polygon_2;

typedef CGAL::Gps_traits_2<Conic_traits_2>     Gps_traits_2;
typedef Gps_traits_2::Polygon_2                Offset_polygon_2;
typedef Gps_traits_2::Polygon_with_holes_2
   Offset_polygon_with_holes_2;

int main ()
{
  // Open the input file.
  std::ifstream    in_file ("spiked.dat");

  if (! in_file.is_open())
  {
```

```
    std::cerr << "Failed to open the input file." << std::endl;
    return (1);
  }

  // Read the input polygon.
  Polygon_2        P;

  in_file >> P;
  in_file.close();

  std::cout << "Read an input polygon with "
            << P.size() << " vertices." << std::endl;

  // Compute the offset polygon.
  Conic_traits_2                traits;
  const Rational                radius = 5;
  Offset_polygon_with_holes_2   offset;
  CGAL::Timer                   timer;

  timer.start();
  offset = offset_polygon_2 (P, radius, traits);
  timer.stop();

  std::cout << "The offset polygon has "
            << offset.outer_boundary().size() << " vertices, "
            << offset.number_of_holes() << " holes." << std::endl;
  std::cout << "Offset computation took "
            << timer.time() << " seconds." << std::endl;
  return (0);
}

#endif
```

File: examples/Minkowski_sum_2/exact_offset.cpp

advanced

Both functions *approximated_offset_2()* and *offset_polygon_2()* also have overloaded versions that accept a decomposition strategy and use the polygon-decomposition approach to compute (or approximate) the offset. These functions are less efficient than their counterparts that employ the convolution approach, and are only included in the package for the sake of completeness.

advanced

24.3.3 Computing Inner Offsets

An operation closely related to the offset computation, is obtaining the *inner offset* of a polygon, or *insetting* it by a given radius. The inset of a polygon P with radius r is the set of points iside P whose distance from the polygon boundary, denoted ∂P, is at least r — namely: $\{p \in P \mid \mathrm{dist}(p, \partial P) \geq r\}$. Note that the resulting set may not be connected in case P is a non-convex polygon that has some narrow components, and thus is characterized by a (possibly empty) set of polygons whose edges are line segments and circular arcs of radius r.

The offset can be computed using the convolution method if we traverse the polygon in a *clockwise* orientation (and not in *counterclockwise* orientation, as done in case of ofsetting a polygon). As in case of the offset functions, the Minkowski-sum package contains two functions for insetting a simple polygon:

The function *approximated_inset_2 (P, r, eps, oi)* accepts a polygon P, an inset radius r and $\varepsilon > 0$. It constructs an approximation for the inset of P by the radius r, with the approximation error bounded by ε. The function returns its output via the output iterator *oi*, whose value-type must be *Gps_circle_segment_traits_2<Kernel> ::Polygon_2* representing the polygons that approximates the inset polygon.

```
#include "ms_rational_nt.h"
#include <CGAL/Lazy_exact_nt.h>
#include <CGAL/Cartesian.h>
#include <CGAL/approximated_offset_2.h>
#include <CGAL/offset_polygon_2.h>
#include <CGAL/Timer.h>
#include <iostream>

typedef CGAL::Lazy_exact_nt<Number_type>           Lazy_exact_nt;

struct Kernel : public CGAL::Cartesian<Lazy_exact_nt> {};
typedef CGAL::Polygon_2<Kernel>                    Polygon_2;

typedef CGAL::Gps_circle_segment_traits_2<Kernel>  Gps_traits_2;
typedef Gps_traits_2::Polygon_2                    Offset_polygon_2;
typedef Gps_traits_2::Polygon_with_holes_2
   Offset_polygon_with_holes_2;

int main ()
{
  // Open the input file.
  std::ifstream    in_file ("tight.dat");

  if (! in_file.is_open())
  {
    std::cerr << "Failed to open the input file." << std::endl;
    return (1);
  }

  // Read the input polygon.
  Polygon_2        P;

  in_file >> P;
  in_file.close();

  std::cout << "Read an input polygon with "
            << P.size() << " vertices." << std::endl;

  // Approximate the offset polygon.
  const Number_type                       radius = 1;
  const double                            err_bound = 0.00001;
  std::list<Offset_polygon_2>             inset_polygons;
  std::list<Offset_polygon_2>::iterator   iit;
  CGAL::Timer                             timer;
```

```
  timer.start();
  approximated_inset_2 (P, radius, err_bound,
                        std::back_inserter (inset_polygons));
  timer.stop();

  std::cout << "The inset comprises "
            << inset_polygons.size() << " polygon(s)." << std::endl;
  for (iit = inset_polygons.begin(); iit != inset_polygons.end(); ++iit)
  {
      std::cout << "    Polygon with "
                << iit->size() << " vertices." << std::endl;
  }
  std::cout << "Inset computation took "
            << timer.time() << " seconds." << std::endl;
  return (0);
}
```

File: examples/Minkowski_sum_2/approx_inset.cpp

Similarly, the function *inset_polygon_2 (P, r, traits, oi)* computes the exact inset of *P* with radius *r*, and returns its output via the given output iterator *oi*. The *traits* parameter should be an instance of an arrangement-traits class that is capable of handling conic arcs in an exact manner, whereas *oi*'s value-type must be *Gps_traits_2< ArrConicTraits>::Polygons_2*.

```
#include <CGAL/basic.h>

#ifndef CGAL_USE_CORE
#include <iostream>
int main ()
{
  std::cout << "Sorry, this example needs CORE ..." << std::endl;
  return (0);
}
#else

#include <CGAL/Cartesian.h>
#include <CGAL/CORE_algebraic_number_traits.h>
#include <CGAL/Arr_conic_traits_2.h>
#include <CGAL/offset_polygon_2.h>
#include <CGAL/Timer.h>
#include <iostream>

typedef CGAL::CORE_algebraic_number_traits     Nt_traits;
typedef Nt_traits::Rational                    Rational;
typedef Nt_traits::Algebraic                   Algebraic;

struct Rat_kernel : public CGAL::Cartesian<Rational> {};
struct Alg_kernel : public CGAL::Cartesian<Algebraic> {};
struct Conic_traits_2 : public CGAL::Arr_conic_traits_2<Rat_kernel,
                                                         Alg_kernel,
                                                         Nt_traits> {};

typedef CGAL::Polygon_2<Rat_kernel>            Polygon_2;
```

```
typedef CGAL::Gps_traits_2<Conic_traits_2>      Gps_traits_2;
typedef Gps_traits_2::Polygon_2                 Offset_polygon_2;

int main ()
{
  // Open the input file.
  std::ifstream    in_file ("tight.dat");

  if (! in_file.is_open())
  {
    std::cerr << "Failed to open the input file." << std::endl;
    return (1);
  }

  // Read the input polygon.
  Polygon_2        P;

  in_file >> P;
  in_file.close();

  std::cout << "Read an input polygon with "
            << P.size() << " vertices." << std::endl;

  // Compute the inner offset of the polygon.
  Conic_traits_2                             traits;
  const Rational                             radius = 1;
  std::list<Offset_polygon_2>                inset_polygons;
  std::list<Offset_polygon_2>::iterator  iit;
  CGAL::Timer                                timer;

  timer.start();
  inset_polygon_2 (P, radius, traits,
                   std::back_inserter (inset_polygons));
  timer.stop();

  std::cout << "The inset comprises "
            << inset_polygons.size() << " polygon(s)." << std::endl;
  for (iit = inset_polygons.begin(); iit != inset_polygons.end(); ++iit)
  {
      std::cout << "    Polygon with "
                << iit->size() << " vertices." << std::endl;
  }
  std::cout << "Inset computation took "
            << timer.time() << " seconds." << std::endl;
  return (0);
}

#endif
```

File: examples/Minkowski_sum_2/exact_inset.cpp

advanced

Unlike the offset functions, there are no overloaded versions of the inset functions that use convex polygon decomposition to compute insets, as this method cannot be easily generalized for inset computations.

advanced

In this context let us mention that there exist overloaded versions of the functions *approximated_offset_2 (P, r, eps)* and *offset_polygon_2 (P, r, traits)* that accept a *polygon with holes P* and computed its offset. This ofset is obtain by taking the outer offset of P's outer boundary, and computing the inner offsets of P's holes. The former polygon defines the output boundary of $P \oplus B_r$, and the latter define the holes within the result.

Acknowledgements

Andreas Fabri and Laurent Rineau helped tracing and solving several bugs in the approximated offset function. They have also suggested a few algorithmic improvements that made their way into version 3.4, yielding a faster approximation scheme.

Planar Minkowski Sums
Reference Manual

Ron Wein

Introduction

This package consist of functions for computing the Minkowksi sum of two polygons in the plane. Namely, given two polygons $P, Q \in \mathbb{R}^2$ it computes $P \oplus Q = \{p + q \mid p \in P, q \in Q\}$.

In addition, the package also includes functions for offsetting a polygon, namely computing its Minkowski sum with a disc of a given radius, and for insetting a polygon (namely computing its inner offset). It is possible to compute the exact representation of the offset (or the inset), or to approximate it with guaranteed error bounds, in order to speed up the computation time.

Functions

Concepts

Classes

24.4 Alphabetical List of Reference Pages

CGAL::minkowski_sum_2

Function

#include <CGAL/minkowski_sum_2.h>

template<class Kernel, class Container>
Polygon_with_holes_2<Kernel,Container>

minkowski_sum_2(Polygon_2<Kernel,Container> P,
Polygon_2<Kernel,Container> Q)

Computes the Minkowski sum $P \oplus Q$ of the two given polygons. The function computes the convolution cycles of the two polygons and extract the regions having positive winding number with respect to these cycles. This method work very efficiently, regardless of whether P and Q are convex or non-convex. Note that as the input polygons may not be convex, their Minkowski sum may not be a simple polygon. The result is therefore represented as a polygon with holes.
Precondition: Both P and Q are simple polygons.

template<class Kernel, class Container, class DecompositionStrategy>
Polygon_with_holes_2<Kernel,Container>

minkowski_sum_2(Polygon_2<Kernel,Container> P,
Polygon_2<Kernel,Container> Q,
DecompositionStrategy decomp)

Computes the Minkowski sum $P \oplus Q$ of the two given polygons. If the input polygons P and Q are not convex, the function decomposes them into convex sub-polygons $P_1, \ldots, P_k$ and $Q_1, \ldots, Q_\ell$ and computes the union of pairwise sub-sums (namely $\bigcup_{i,j} (P_i \oplus Q_j)$). The decomposition is performed using the given decomposition strategy *decomp*, which must be an instance of a class that models the concept *PolygonConvexDecomposition*. Note that as the input polygons may not be convex, their Minkowski sum may not be a simple polygon. The result is therefore represented as a polygon with holes.
Precondition: Both P and Q are simple polygons.

Function

CGAL::approximated_offset_2

#include <CGAL/approximated_offset_2.h>

template<class Kernel, class Container>
typename Gps_circle_segment_traits_2<Kernel>::Polygon_with_holes_2

approximated_offset_2(Polygon_2<Kernel, Container> P,
typename Kernel::FT r,
double eps)

Provides a guaranteed approximation of the offset of the given polygon P by a given radius r — namely, the function computes the Minkowski sum $P \oplus B_r$, where B_r is a disc of radius r centered at the origin. The function actually outputs a set S that contains the Minkowski sum, such that the approximation error is bounded by *eps*. Note that as the input polygon may not be convex, its offset may not be a simple polygon. The result is therefore represented as a polygon with holes, whose edges are either line segments or circular arcs.
Precondition: P is a simple polygon.

template<class Kernel, class Container>
typename Gps_circle_segment_traits_2<Kernel>::Polygon_with_holes_2

approximated_offset_2(Polygon_with_holes_2<Kernel, Container> wh,
typename Kernel::FT r,
double eps)

Provides a guaranteed approximation of offset the given polygon with holes *pwh* by a given radius r, such that the approximation error is bounded by *eps*. It does so by offsetting outer boundary of *pwh* and insetting its holes. The result is represented as a generalized polygon with holes, such that the edges of the polygon correspond to line segment and circular arcs.
Precondition: *pwh* is *not* unbounded (it has a valid outer boundary).

template<class Kernel, class Container, class DecompositionStrategy>
typename Gps_circle_segment_traits_2<Kernel>::Polygon_with_holes_2

approximated_offset_2(Polygon_2<Kernel, Container> P,
typename Kernel::FT r,
double eps,

DecompositionStrategy decomp)

Provides a guaranteed approximation of the offset of the given polygon P by a radius r, as described above. If the input polygon P is not convex, the function decomposes it into convex sub-polygons $P_1, \ldots, P_k$ and computes the union of the sub-offsets (namely $\bigcup_i (P_i \oplus B_r)$). The decomposition is performed using the given decomposition strategy *decomp*, which must be an instance of a class that models the concept *PolygonConvexDecomposition*.
Precondition: P is a simple polygon.

CGAL::approximated_inset_2

#include <CGAL/approximated_offset_2.h>

template<class Kernel, class Container, class OutputIterator>
OutputIterator approximated_inset_2(Polygon_2<Kernel, Container> pgn, typename Kernel::FT r, double eps, OutputIterator oi)

Provides a guaranteed approximation of the inset, or inner offset, of the given polygon P by a given radius r. Namely, the function computes the set of points inside the polygon whose distance from P's boundary is at least r: $\{p \in P \mid \mathrm{dist}(p, \partial P) \geq r\}$, with the approximation error bounded by *eps*. Note that as the input polygon may not be convex, its inset may comprise several disconnected components. The result is therefore represented as a sequence of generalized polygons, whose edges are either line segments or circular arcs. The output sequence is returned via the output iterator *oi*, whose value-type must be *Gps_circle_segment_traits_2< Kernel>::Polygon_2*.
Precondition: P is a simple polygon.

CGAL::offset_polygon_2

#include <CGAL/offset_polygon_2.h>

template<class ConicTraits, class Container>
typename Gps_traits_2<ConicTraits>::Polygon_with_holes_2

offset_polygon_2(Polygon_2<typename ConicTraits::Rat_kernel, Container> P,
typename ConicTraits::Rat_kernel::FT r,
ConicTraits traits)

Computes the offset of the given polygon P by a given radius r — namely, the function computes the Minkowski sum $P \oplus B_r$, where B_r is a disc of radius r centered at the origin. Note that as the input polygon may not be convex, its offset may not be a simple polygon. The result is therefore represented as a generalized polygon with holes, such that the edges of the polygon correspond to line segments and circular arcs, both are special types of conic arcs, as represented by the *traits* class.
Precondition: P is a simple polygon.

template<class ConicTraits, class Container>
typename Gps_traits_2<ConicTraits>::Polygon_with_holes_2

offset_polygon_2(Polygon_with_holes_2<typename ConicTraits::Rat_kernel, Container> pwh,
typename ConicTraits::Rat_kernel::FT r,
ConicTraits traits)

Computes the offset of the given polygon with holes *pwh* by a given radius r. It does so by offsetting outer boundary of *pwh* and insetting its holes. The result is represented as a generalized polygon with holes, such that the edges of the polygon correspond to line segments and circular arcs, both are special types of conic arcs, as represented by the *traits* class.
Precondition: *pwh* is *not* unbounded (it has a valid outer boundary).

template<class ConicTraits, class Container, class DecompositionStrategy>
typename Gps_traits_2<ConicTraits>::Polygon_with_holes_2

offset_polygon_2(Polygon_2<typename ConicTraits::Rat_kernel, Container> P,
typename ConicTraits::Rat_kernel::FT r,
DecompositionStrategy decomp,

Function

ConicTraits traits)

Computes the exact representation of the offset of the given polygon P by a radius r, as described above. If P is not convex, the function decomposes it into convex sub-polygons $P_1, \ldots, P_k$ and computes the union of sub-offsets (namely $\bigcup_i (P_i \oplus B_r)$). The decomposition is performed using the given decomposition strategy *decomp*, which must be an instance of a class that models the concept *PolygonConvexDecomposition*.
Precondition: P is a simple polygon.

CGAL::inset_polygon_2

#include <CGAL/offset_polygon_2.h>

```
template<class ConicTraits, class Container, class OutputIterator>
OutputIterator    inset_polygon_2( Polygon_2<typename ConicTraits::Rat_kernel, Container> P,
                                   typename ConicTraits::Rat_kernel::FT r,
                                   ConicTraits traits,
                                   OutputIterator oi)
```

Computes the inset, or inner offset, of the given polygon P by a given radius r — namely, the function computes the set of points inside the polygon whose distance from P's boundary is at least r: $\{p \in P \mid \mathrm{dist}(p, \partial P) \geq r\}$. Note that as the input polygon may not be convex, its inset may comprise several disconnected components. The result is therefore represented as a sequence of generalized polygons, such that the edges of each polygon correspond to line segments and circular arcs, both are special types of conic arcs, as represented by the *traits* class. The output sequence is returned via the output iterator *oi*, whose value-type must be *Gps_traits_2< ConicTraits>::Polygon_2*.
Precondition: P is a simple polygon.

Function

PolygonConvexDecomposition_2

A model of the PolygonConvexDecomposition_2 concept is capable of decomposing an input polygon P into a set of convex sub-polygons $P_1, \ldots, P_k$, such that $\cup_{i=1}^{k} P_k = P$.

Types

PolygonConvexDecomposition_2:: Kernel	the geometric kernel type.
PolygonConvexDecomposition_2:: Point_2	the point type, used to represent polygon vertices.
PolygonConvexDecomposition_2:: Polygon_2	the polygon type.

Creation

PolygonConvexDecomposition_2 decomp;	default constructor.

Operations

template <class OutputIterator>
OutputIterator *decomp(Polygon_2 P, OutputIterator oi)*

decomposes the input polygon P into convex sub-polygons, and writes them to the output iterator *oi*. The value-type of the output iterator must be *Polygon_2*. The function returns a past-the-end iterator for the convex sub-polygons.

Has Models

Small_side_angle_bisector_decomposition_2<Kernel,Container> (page 1771)
Optimal_convex_decomposition_2<Kernel,Container> (page 1772)
Hertel_Mehlhorn_convex_decomposition_2<Kernel,Container> (page 1773)
Greene_convex_decomposition_2<Kernel,Container> (page 1774)

CGAL::Small_side_angle_bisector_decomposition_2<Kernel,Container>

Class

Definition

The *Small_side_angle_bisector_decomposition_2<Kernel,Container>* class implements a simple yet efficient heuristic for decomposing an input polygon into convex sub-polygons. It is based on the algorithm suggested by Flato and Halperin [FH00], but without introducing Steiner points. The algorithm operates in two major steps. In the first step, it tries to subdivide the polygon by connect two reflex vertices with an edge. When this is not possible any more, it eliminates the reflex vertices one by one by connecting them to other convex vertices, such that the new edge best approximates the angle bisector of the reflex vertex. The algorithm operates in $O(n^2)$ time an takes $O(n)$ space at the worst case, where n is the size of the input polygon.

The *Polygon_2* type defined by the class is simply *Polygon_2<Kernel,Container>*. The *Container* parameter is by default *std::vector<typename Kernel::Point_2>*.

#include <CGAL/Small_side_angle_bisector_decomposition_2.h>

Is Model for the Concepts

PolygonConvexDecomposition_2

Class

CGAL::Optimal_convex_decomposition_2<Kernel,Container>

Definition

The *Optimal_convex_decomposition_2<Kernel,Container>* class provides an implementation of Greene's dynamic programming algorithm for optimal decomposition of a polygon into convex sub-polygons [Gre83]. Note that this algorithm requires $O(n^4)$ time and $O(n^3)$ space in the worst case, where n is the size of the input polygon.

The *Polygon_2* type defined by the class is simply *Polygon_2<Kernel,Container>*. The *Container* parameter is by default *std::vector<typename Kernel::Point_2>*.

#include <CGAL/Polygon_convex_decomposition_2.h>

Is Model for the Concepts

PolygonConvexDecomposition_2

See Also

CGAL::optimal_convex_partition_2(page 1646)

CGAL::Hertel_Mehlhorn_convex_decomposition_2<Kernel,Container>

Class

Definition

The *Hertel_Mehlhorn_convex_decomposition_2<Kernel,Container>* class implements the approximation algorithm of Hertel and Mehlhorn for decomposing a polygon into convex sub-polygons [HM83]. This algorithm constructs a triangulation of the input polygon and proceeds by removing unnecessary triangulation edges. Given the triangulation, the algorithm requires $O(n)$ time and space to construct a convex decomposition (where n is the size of the input polygon), whose size is guaranteed to be no more than four times the size of the optimal decomposition.

The *Polygon_2* type defined by the class is simply *Polygon_2<Kernel,Container>*. The *Container* parameter is by default *std::vector<typename Kernel::Point_2>*.

#include <CGAL/Polygon_convex_decomposition_2.h>

Is Model for the Concepts

PolygonConvexDecomposition_2

See Also

CGAL::approx_convex_partition_2(page 1630)

CGAL::Greene_convex_decomposition_2<Kernel,Container>

Definition

The *Greene_convex_decomposition_2<Kernel,Container>* class implements the approximation algorithm of Greene for the decomposition of an input polygon into convex sub-polygons [Gre83]. This algorithm takes $O(n \log n)$ time and $O(n)$ space, where n is the size of the input polygon, and outputs a decomposition whose size is guaranteed to be no more than four times the size of the optimal decomposition.

The *Polygon_2* type defined by the class is simply *Polygon_2<Kernel,Container>*. The *Container* parameter is by default *std::vector<typename Kernel::Point_2>*.

#include <CGAL/Polygon_convex_decomposition_2.h>

Is Model for the Concepts

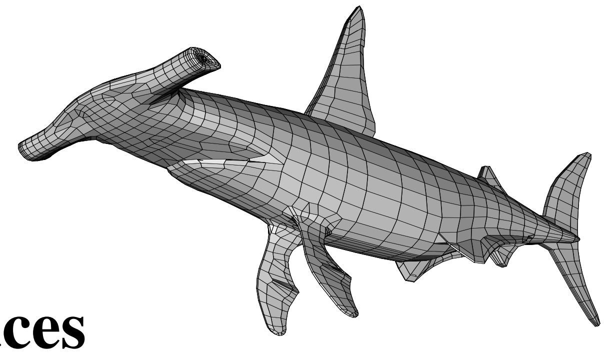
PolygonConvexDecomposition_2

See Also

CGAL::greene_approx_convex_partition_2(page 1636)

Part VII

Cell Complexes and Polyhedra



Chapter 25

3D Polyhedral Surfaces

Lutz Kettner

Contents

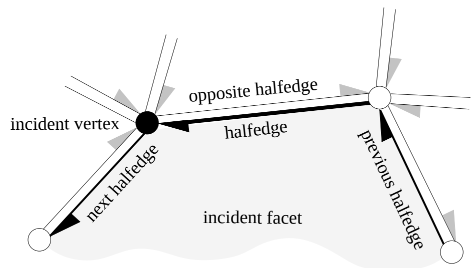
25.1 Introduction

Polyhedral surfaces in three dimensions are composed of vertices, edges, facets and an incidence relationship on them. The organization beneath is a halfedge data structure, which restricts the class of representable surfaces to orientable 2-manifolds – with and without boundary. If the surface is closed we call it a *polyhedron*, for example, see the above model of a hammerhead:

The polyhedral surface is realized as a container class that manages vertices, halfedges, facets with their incidences, and that maintains the combinatorial integrity of them. It is based on the highly flexible design of the halfedge data structure, see the introduction in Chapter 26 and [Ket99]. However, the polyhedral surface can be used and understood without knowing the underlying design. Some of the examples in this chapter introduce also gradually into first applications of this flexibility.

25.2 Definition

A polyhedral surface *CGAL::Polyhedron_3<PolyhedronTraits_3>* in three dimensions consists of vertices V, edges E, facets F and an incidence relation on them. Each edge is represented by two halfedges with opposite orientations. The incidences stored with a halfedge are illustrated in the following figure:



Vertices represent points in space. Edges are straight line segments between two endpoints. Facets are planar polygons without holes. Facets are defined by the circular sequence of halfedges along their boundary. The polyhedral surface itself can have holes (with at least two facets surrounding it since a single facet cannot have a hole). The halfedges along the boundary of a hole are called *border halfedges* and have no incident facet. An edge is a *border edge* if one of its halfedges is a border halfedge. A surface is *closed* if it contains no border halfedges. A closed surface is a boundary representation for polyhedra in three dimensions. The convention is that the halfedges are oriented counterclockwise around facets as seen from the outside of the polyhedron. An implication is that the halfedges are oriented clockwise around the vertices. The notion of the solid side of a facet as defined by the halfedge orientation extends to polyhedral surfaces with border edges although they do not define a closed object. If normal vectors are considered for the facets, normals point outwards (following the right-hand rule).

The strict definition can be found in [Ket99]. One implication of this definition is that the polyhedral surface is always an orientable and oriented 2-manifold with border edges, i.e., the neighborhood of each point on the polyhedral surface is either homeomorphic to a disc or to a half disc, except for vertices where many holes and surfaces with boundary can join. Another implication is that the smallest representable surface avoiding self intersections is a triangle (for polyhedral surfaces with border edges) or a tetrahedron (for polyhedra). Boundary representations of orientable 2-manifolds are closed under Euler operations. They are extended with operations that create or close holes in the surface.

Other intersections besides the incidence relation are not allowed. However, this is not automatically verified in the operations, since self intersections are not easy to check efficiently. *CGAL::Polyhedron_3< PolyhedronTraits_3>* does only maintain the combinatorial integrity of the polyhedral surface (using Euler operations) and does not consider the coordinates of the points or any other geometric information.

CGAL::Polyhedron_3<PolyhedronTraits_3> can represent polyhedral surfaces as well as polyhedra. The interface is designed in such a way that it is easy to ignore border edges and work only with polyhedra.

25.3 Example Programs

The polyhedral surface is based on the highly flexible design of the halfedge data structure. Examples for this flexibility can be found in Section 25.5 and in Section 26.3. This design is not a prerequisite to understand the following examples. See also the Section 25.6 below for some advanced examples.

25.3.1 First Example Using Defaults

The first example instantiates a polyhedron using a kernel as traits class. It creates a tetrahedron and stores the reference to one of its halfedges in a *Halfedge_handle*. Handles, also know as *trivial iterators*, are used to keep references to halfedges, vertices, or facets for future use. There is also a *Halfedge_iterator* type for enumerating halfedges. Such an iterator type can be used wherever a handle is required. Respective *Halfedge_const_handle* and *Halfedge_const_iterator* for a constant polyhedron and similar handles and iterators with *Vertex_* and *Facet_* prefix are provided too.

The example continues with a test if the halfedge actually refers to a tetrahedron. This test checks the connected component referred to by the halfedge *h* and not the polyhedral surface as a whole. This examples works only on the combinatorial level of a polyhedral surface. The next example adds the geometry.

```
#include <CGAL/Simple_cartesian.h>
#include <CGAL/Polyhedron_3.h>

typedef CGAL::Simple_cartesian<double>     Kernel;
typedef CGAL::Polyhedron_3<Kernel>         Polyhedron;
typedef Polyhedron::Halfedge_handle        Halfedge_handle;

int main() {
    Polyhedron P;
    Halfedge_handle h = P.make_tetrahedron();
    if ( P.is_tetrahedron(h))
        return 0;
    return 1;
}
```

File: `examples/Polyhedron/polyhedron_prog_simple.cpp`

25.3.2 Example with Geometry in Vertices

We add geometry to the our construction of a tetrahedron. Four points are passed as arguments to the construction. This example demonstrates in addition the use of the vertex iterator and the access to the point in the vertices. Note the natural access notation *v->point()*. Similarly, all information stored in a vertex, halfedge, and facet can be accessed with a member function given a handle or iterator. For example, given a halfedge handle *h* we can write *h->next()* to get a halfedge handle to the next halfedge, *h->opposite()* for the opposite halfedge, *h->vertex()* for the incident vertex at the tip of *h*, and so on. The output of the program will be "`1 0 0\n0 1 0\n0 0 1\n0 0 0\n`".

```
#include <CGAL/Simple_cartesian.h>
#include <CGAL/Polyhedron_3.h>
#include <iostream>

typedef CGAL::Simple_cartesian<double>     Kernel;
typedef Kernel::Point_3                    Point_3;
typedef CGAL::Polyhedron_3<Kernel>         Polyhedron;
typedef Polyhedron::Vertex_iterator        Vertex_iterator;

int main() {
    Point_3 p( 1.0, 0.0, 0.0);
    Point_3 q( 0.0, 1.0, 0.0);
```

```
    Point_3 r( 0.0, 0.0, 1.0);
    Point_3 s( 0.0, 0.0, 0.0);

    Polyhedron P;
    P.make_tetrahedron( p, q, r, s);
    CGAL::set_ascii_mode( std::cout);
    for ( Vertex_iterator v = P.vertices_begin(); v != P.vertices_end();
   ++v)
        std::cout << v->point() << std::endl;
    return 0;
}
```

File: examples/Polyhedron/polyhedron_prog_tetra.cpp

The polyhedron offers a point iterator for convenience. The above `for` loop simplifies to a single statement by using *std::copy* and the ostream iterator adaptor.

```
std::copy( P.points_begin(), P.points_end(),
           std::ostream_iterator<Point_3>(std::cout,"\n"));
```

25.3.3 Example for Affine Transformation

An affine transformation *A* can act as a functor transforming points and a point iterator is conveniently defined for polyhedral surfaces. So, assuming we want only the point coordinates of a polyhedron *P* transformed, *std::transform* does the job in a single line.

```
std::transform( P.points_begin(), P.points_end(), P.points_begin(), A);
```

25.3.4 Example Computing Plane Equations

The polyhedral surface has already provisions to store a plane equation for each facet. However, it does not provide a function to compute plane equations.

This example computes the plane equations of a polyhedral surface. The actual computation is implemented in the `compute_plane_equations` function. Depending on the arithmetic (exact/inexact) and the shape of the facets (convex/non-convex) different methods are useful. We assume here strictly convex facets and exact arithmetic. In our example a homogeneous representation with `int` coordinates is sufficient. The four plane equations of the tetrahedron are the output of the program.

```
#include <CGAL/Homogeneous.h>
#include <CGAL/Polyhedron_3.h>
#include <iostream>
#include <algorithm>

struct Plane_equation {
    template <class Facet>
    typename Facet::Plane_3 operator()( Facet& f) {
        typename Facet::Halfedge_handle h = f.halfedge();
```

```
        typedef typename Facet::Plane_3  Plane;
        return Plane( h->vertex()->point(),
                      h->next()->vertex()->point(),
                      h->next()->next()->vertex()->point());
    }
};

typedef CGAL::Homogeneous<int>      Kernel;
typedef Kernel::Point_3             Point_3;
typedef Kernel::Plane_3             Plane_3;
typedef CGAL::Polyhedron_3<Kernel>  Polyhedron;

int main() {
    Point_3 p( 1, 0, 0);
    Point_3 q( 0, 1, 0);
    Point_3 r( 0, 0, 1);
    Point_3 s( 0, 0, 0);
    Polyhedron P;
    P.make_tetrahedron( p, q, r, s);
    std::transform( P.facets_begin(), P.facets_end(), P.planes_begin(),
                    Plane_equation());
    CGAL::set_pretty_mode( std::cout);
    std::copy( P.planes_begin(), P.planes_end(),
               std::ostream_iterator<Plane_3>( std::cout, "\n"));
    return 0;
}
```

File: examples/Polyhedron/polyhedron_prog_planes.cpp

25.3.5 Example with a Vector Instead of a List Representation

The polyhedron class template has actually four parameters, where three of them have default values. Using the default values explicitly in our examples above for three parameter—ignoring the fourth parameter, which would be a standard allocator for container class— the definition of a polyhedron looks like:

```
typedef CGAL::Polyhedron_3< Traits,
                            CGAL::Polyhedron_items_3,
                            CGAL::HalfedgeDS_default>      Polyhedron;
```

The *CGAL::Polyhedron_items_3* class contains the types used for vertices, edges, and facets. The *CGAL::HalfedgeDS_default* class defines the halfedge data structure used, which is a list-based representation in this case. An alternative is a vector-based representation. Using a vector provides random access for the elements in the polyhedral surface and is more space efficient, but elements cannot be deleted arbitrarily. Using a list allows arbitrary deletions, but provides only bidirectional iterators and is less space efficient. The following example creates again a tetrahedron with given points, but in a vector-based representation.

The vector-based representation resizes automatically if the reserved capacity is not sufficient for the new items created. Upon resizing all handles, iterators, and circulators become invalid. Their correct update in the halfedge data structure is costly, thus it is advisable to reserve enough space in advance as indicated with the alternative constructor in the comment.

advanced

Note that the polyhedron and not the underlying halfedge data structure triggers the resize operation, since the resize operation requires some preconditions, such as valid incidences, to be fulfilled that only the polyhedron can guarantee.

advanced

```
#include <CGAL/Cartesian.h>
#include <CGAL/HalfedgeDS_vector.h>
#include <CGAL/Polyhedron_3.h>
#include <iostream>

typedef CGAL::Cartesian<double>                              Kernel;
typedef Kernel::Point_3                                      Point_3;
typedef CGAL::Polyhedron_3< Kernel,
                            CGAL::Polyhedron_items_3,
                            CGAL::HalfedgeDS_vector>   Polyhedron;

int main() {
    Point_3 p( 1.0, 0.0, 0.0);
    Point_3 q( 0.0, 1.0, 0.0);
    Point_3 r( 0.0, 0.0, 1.0);
    Point_3 s( 0.0, 0.0, 0.0);

    Polyhedron P;    // alternative constructor: Polyhedron P(4,12,4);
    P.make_tetrahedron( p, q, r, s);
    CGAL::set_ascii_mode( std::cout);
    std::copy( P.points_begin(), P.points_end(),
               std::ostream_iterator<Point_3>( std::cout, "\n"));
    return 0;
}
```

File: examples/Polyhedron/polyhedron_prog_vector.cpp

25.3.6 Example with Circulator Writing Object File Format (OFF)

We create a tetrahedron and write it to *std::cout* using the Object File Format (OFF) [Phi96]. This example makes use of STL algorithms (*std::copy*, *std::distance*), STL *std::ostream_iterator*, and CGAL circulators. The polyhedral surface provides convenient circulators for the counterclockwise circular sequence of halfedges around a facet and the clockwise circular sequence of halfedges around a vertex.

However, the computation of the vertex index in the inner loop of the facet output is not advisable with the *std::distance* function, since it takes linear time for non random-access iterators, which leads to quadratic runtime. For better runtime the vertex index needs to be stored separately and computed once before writing the facets. It can be stored, for example, in the vertex itself or in a hash-structure. See also the following Section 25.4 for file I/O.

```
#include <CGAL/Simple_cartesian.h>
#include <CGAL/Polyhedron_3.h>
#include <iostream>

typedef CGAL::Simple_cartesian<double>                     Kernel;
```

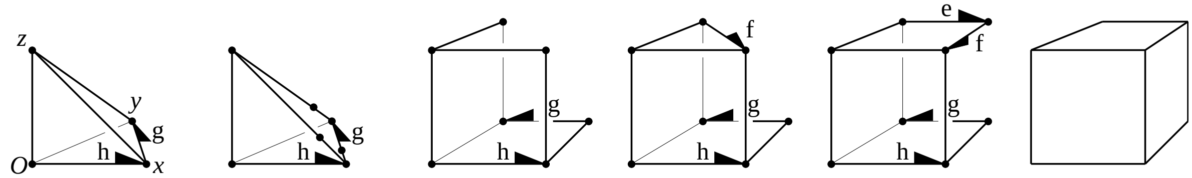
```
typedef Kernel::Point_3                                Point_3;
typedef CGAL::Polyhedron_3<Kernel>                     Polyhedron;
typedef Polyhedron::Facet_iterator                     Facet_iterator;
typedef Polyhedron::Halfedge_around_facet_circulator
   Halfedge_facet_circulator;

int main() {
    Point_3 p( 0.0, 0.0, 0.0);
    Point_3 q( 1.0, 0.0, 0.0);
    Point_3 r( 0.0, 1.0, 0.0);
    Point_3 s( 0.0, 0.0, 1.0);

    Polyhedron P;
    P.make_tetrahedron( p, q, r, s);

    // Write polyhedron in Object File Format (OFF).
    CGAL::set_ascii_mode( std::cout);
    std::cout << "OFF" << std::endl << P.size_of_vertices() << ' '
              << P.size_of_facets() << " 0" << std::endl;
    std::copy( P.points_begin(), P.points_end(),
               std::ostream_iterator<Point_3>( std::cout, "\n"));
    for (  Facet_iterator i = P.facets_begin(); i != P.facets_end();
   ++i) {
        Halfedge_facet_circulator j = i->facet_begin();
        // Facets in polyhedral surfaces are at least triangles.
        CGAL_assertion( CGAL::circulator_size(j) >= 3);
        std::cout << CGAL::circulator_size(j) << ' ';
        do {
            std::cout << ' ' << std::distance(P.vertices_begin(),
   j->vertex());
        } while ( ++j != i->facet_begin());
        std::cout << std::endl;
    }
    return 0;
}
```

File: examples/Polyhedron/polyhedron_prog_off.cpp

25.3.7 Example Using Euler Operators to Build a Cube

Euler operators are the natural way of modifying polyhedral surfaces. We provide a set of operations for polyhedra: *split_facet()*, *join_facet()*, *split_vertex()*, *join_vertex()*, *split_loop()*, and *join_loop()*. We add further convenient operators, such as *split_edge()*. However, they could be implemented using the six operators above. Furthermore, we provide more operators to work with polyhedral surfaces with border edges, for example, creating and deleting holes. We refer to the references manual for the definition and illustrative figures of the Euler operators.

The following example implements a function that appends a unit cube to a polyhedral surface. To keep track of the different steps during the creation of the cube a sequence of sketches might help with labels for the different handles that occur in the program code. The following Figure shows six selected steps from the creation sequence. These steps are also marked in the program code.



(a) **(b)** **(c)** **(d)** **(e)** **(f)**

```
#include <CGAL/Simple_cartesian.h>
#include <CGAL/Polyhedron_3.h>
#include <iostream>

template <class Poly>
typename Poly::Halfedge_handle make_cube_3( Poly& P) {
    // appends a cube of size [0,1]^3 to the polyhedron P.
    CGAL_precondition( P.is_valid());
    typedef typename Poly::Point_3         Point;
    typedef typename Poly::Halfedge_handle Halfedge_handle;
    Halfedge_handle h = P.make_tetrahedron( Point( 1, 0, 0),
                                            Point( 0, 0, 1),
                                            Point( 0, 0, 0),
                                            Point( 0, 1, 0));
    Halfedge_handle g = h->next()->opposite()->next();             //
   Fig. (a)
    P.split_edge( h->next());
    P.split_edge( g->next());
    P.split_edge( g);                                               //
   Fig. (b)
    h->next()->vertex()->point()     = Point( 1, 0, 1);
    g->next()->vertex()->point()     = Point( 0, 1, 1);
    g->opposite()->vertex()->point() = Point( 1, 1, 0);             //
   Fig. (c)
    Halfedge_handle f = P.split_facet( g->next(),
                                       g->next()->next()->next()); //
   Fig. (d)
    Halfedge_handle e = P.split_edge( f);
    e->vertex()->point() = Point( 1, 1, 1);                         //
   Fig. (e)
    P.split_facet( e, f->next()->next());                           //
   Fig. (f)
    CGAL_postcondition( P.is_valid());
    return h;
}

typedef CGAL::Simple_cartesian<double>     Kernel;
typedef CGAL::Polyhedron_3<Kernel>         Polyhedron;
typedef Polyhedron::Halfedge_handle        Halfedge_handle;

int main() {
    Polyhedron P;
    Halfedge_handle h = make_cube_3( P);
    return (P.is_tetrahedron(h) ? 1 : 0);
}
```

File: `examples/Polyhedron/polyhedron_prog_cube.cpp`

25.4 File I/O

Simple file I/O for polyhedral surfaces is already provided in the library. The file I/O considers so far only the topology of the surface and its point coordinates. It ignores a possible plane equation or any user-added attributes, such as color.

The default file format supported in CGAL for output as well as for input is the Object File Format, OFF, with file extension `.off`, which is also understood by Geomview [Phi96]. For OFF an ASCII and a binary format exist. The format can be selected with the CGAL modifiers for streams, *set_ascii_mode* and *set_binary_mode* respectively. The modifier *set_pretty_mode* can be used to allow for (a few) structuring comments in the output. Otherwise, the output would be free of comments. The default for writing is ASCII without comments. Both, ASCII and binary format, can be read independent of the stream setting. Since this file format is the default format, iostream operators are provided for it.

#include <CGAL/IO/Polyhedron_iostream.h>

template <class PolyhedronTraits_3>
ostream& *ostream& out << CGAL::Polyhedron_3<PolyhedronTraits_3> P*

template <class PolyhedronTraits_3>
istream& *istream& in >> CGAL::Polyhedron_3<PolyhedronTraits_3>& P*

Additional formats supported for writing are OpenInventor (`.iv`) [Wer94], VRML 1.0 and 2.0 (`.wrl`) [BPP95, VRM96, HW96], and Wavefront Advanced Visualizer object format (`.obj`). Another convenient output function writes a polyhedral surface to a Geomview process spawned from the CGAL program. These output functions are provided as stream operators, now acting on the stream type of the respective format.

#include <CGAL/IO/Polyhedron_inventor_ostream.h>
#include <CGAL/IO/Polyhedron_VRML_1_ostream.h>
#include <CGAL/IO/Polyhedron_VRML_2_ostream.h>
#include <CGAL/IO/Polyhedron_geomview_ostream.h>

template <class PolyhedronTraits_3>
Inventor_ostream& *Inventor_ostream& out << CGAL::Polyhedron_3<PolyhedronTraits_3> P*

template <class PolyhedronTraits_3>
VRML_1_ostream& *VRML_1_ostream& out << CGAL::Polyhedron_3<PolyhedronTraits_3> P*

template <class PolyhedronTraits_3>
VRML_2_ostream& *VRML_2_ostream& out << CGAL::Polyhedron_3<PolyhedronTraits_3> P*

template <class PolyhedronTraits_3>
Geomview_stream& *Geomview_stream& out << CGAL::Polyhedron_3<PolyhedronTraits_3> P*

All these file formats have in common that they represent a surface as a set of facets. Each facet is a list of indices pointing into a set of vertices. Vertices are represented as coordinate triples. The file I/O for polyhedral surfaces *CGAL::Polyhedron_3* imposes certain restrictions on these formats. They must represent a permissible polyhedral surface, e.g., a 2-manifold and no isolated vertices, see Section 25.1.

Some example programs around the different file formats are provided in the distribution under examples/Polyhedron_IO/ and demo/Polyhedron_IO/. We show an example converting OFF input into VRML 1.0 output.

```
// examples/Polyhedron_IO/polyhedron2vrml.cpp
// ----------------------------------------

#include <CGAL/Simple_cartesian.h>
#include <CGAL/Polyhedron_3.h>
#include <CGAL/IO/Polyhedron_iostream.h>
#include <CGAL/IO/Polyhedron_VRML_1_ostream.h>
#include <iostream>

typedef CGAL::Simple_cartesian<double> Kernel;
typedef CGAL::Polyhedron_3<Kernel>     Polyhedron;

int main() {
    Polyhedron P;
    std::cin >> P;
    CGAL::VRML_1_ostream out( std::cout);
    out << P;
    return ( std::cin && std::cout) ? 0 : 1;
}
```

25.5 Extending Vertices, Halfedges, and Facets

In Section 25.3.5 we have seen how to change the default list representation

```
typedef CGAL::Polyhedron_3< Traits,
                            CGAL::Polyhedron_items_3,
                            CGAL::HalfedgeDS_default>      Polyhedron;
```

to a vector based representation of the underlying halfedge data structure. Now we want to look a bit closer at the second template argument, Polyhedron_items_3, that specifies what kind of vertex, halfedge, and facet is used. The implementation of Polyhedron_items_3 looks a bit involved with nested wrapper class templates. But ignoring this technicality, what remains are three local typedefs that define the Vertex, the Halfedge, and the Face for the polyhedral surface. Note that we use here Face instead of facet. Face is the term used for the halfedge data structure. Only the top layer of the polyhedral surface gives alias names renaming face to facet.

```
class Polyhedron_items_3 {
public:
    template < class Refs, class Traits>
    struct Vertex_wrapper {
        typedef typename Traits::Point_3 Point;
        typedef CGAL::HalfedgeDS_vertex_base<Refs, CGAL::Tag_true, Point>
     Vertex;
    };
    template < class Refs, class Traits>
    struct Halfedge_wrapper {
        typedef CGAL::HalfedgeDS_halfedge_base<Refs>
     Halfedge;
    };
```

```
    template < class Refs, class Traits>
    struct Face_wrapper {
        typedef typename Traits::Plane_3 Plane;
        typedef CGAL::HalfedgeDS_face_base<Refs, CGAL::Tag_true, Plane>
    Face;
    };
};
```

If we look up in the reference manual the definitions of the three classes used in the typedefs, we will see the confirmation that the default polyhedron uses all supported incidences, a point in the vertex class, and a plane equation in the face class. Note how the wrapper class provides two template parameters, `Refs`, which we discuss a bit later, and `Traits`, which is the geometric traits class used by the polyhedral surface and which provides us here with the types for the point and the plane equation.

Using this example code we can write our own items class. Instead, we illustrate an easier way if we only want to exchange one class. We use a simpler face without the plane equation but with a color attribute added. To simplify the creation of a vertex, halfedge, or face class, it is always recommended to derive from one of the given base classes. Even if the base class would contain no data it would provide convenient type definitions. So, we derive from the base class, repeat the mandatory constructors if necessary—which is not the case for faces but would be for vertices—and add the color attribute.

```
template <class Refs>
struct My_face : public CGAL::HalfedgeDS_face_base<Refs> {
    CGAL::Color color;
};
```

The new items class is derived from the old items class and the wrapper containing the face typedef gets overridden. Note that the name of the wrapper and its template parameters are fixed. They cannot be changed even if, as in this example, a template parameter is not used.

```
struct My_items : public CGAL::Polyhedron_items_3 {
    template <class Refs, class Traits>
    struct Face_wrapper {
        typedef My_face<Refs> Face;
    };
};
```

When we use our new items class with the polyhedral surface, our new face class is used in the halfedge data structure and the color attribute is available in the type `Polyhedron::Facet`. However, `Polyhedron::Facet` is not the same type as our local face typedef for `My_face`, but it is derived therefrom. Thus, everything that we put in the local face type except constructors is then available in the `Polyhedron::Facet` type. For more details, see the Chapter 26 on the halfedge data structure design.

Pulling all pieces together, the full example program illustrates how easy the color attribute can be accessed once it is defined.

```
#include <CGAL/Simple_cartesian.h>
#include <CGAL/IO/Color.h>
#include <CGAL/Polyhedron_3.h>

// A face type with a color member variable.
template <class Refs>
struct My_face : public CGAL::HalfedgeDS_face_base<Refs> {
    CGAL::Color color;
```

```
};

// An items type using my face.
struct My_items : public CGAL::Polyhedron_items_3 {
    template <class Refs, class Traits>
    struct Face_wrapper {
        typedef My_face<Refs> Face;
    };
};

typedef CGAL::Simple_cartesian<double>        Kernel;
typedef CGAL::Polyhedron_3<Kernel, My_items>  Polyhedron;
typedef Polyhedron::Halfedge_handle           Halfedge_handle;

int main() {
    Polyhedron P;
    Halfedge_handle h = P.make_tetrahedron();
    h->facet()->color = CGAL::RED;
    return 0;
}
```

File: examples/Polyhedron/polyhedron_prog_color.cpp

We come back to the first template parameter, Refs, of the wrapper classes. This parameter provides us with local types that allow us to make further references between vertices, halfedges, and facets, which have not already been prepared for in the current design. These local types are Vertex_handle, Halfedge_handle, Face_handle, and there respective ..._const_handle. We add now a new vertex reference to a face class as follows. Encapsulation and access functions could be added for a more thorough design, but we omit that here for the sake of brevity. The integration of the face class with the items class works as illustrated above.

```
template <class Refs>
struct My_face : public CGAL::HalfedgeDS_face_base<Refs> {
    typedef typename Refs::Vertex_handle Vertex_handle;
    Vertex_handle vertex_ref;
};
```

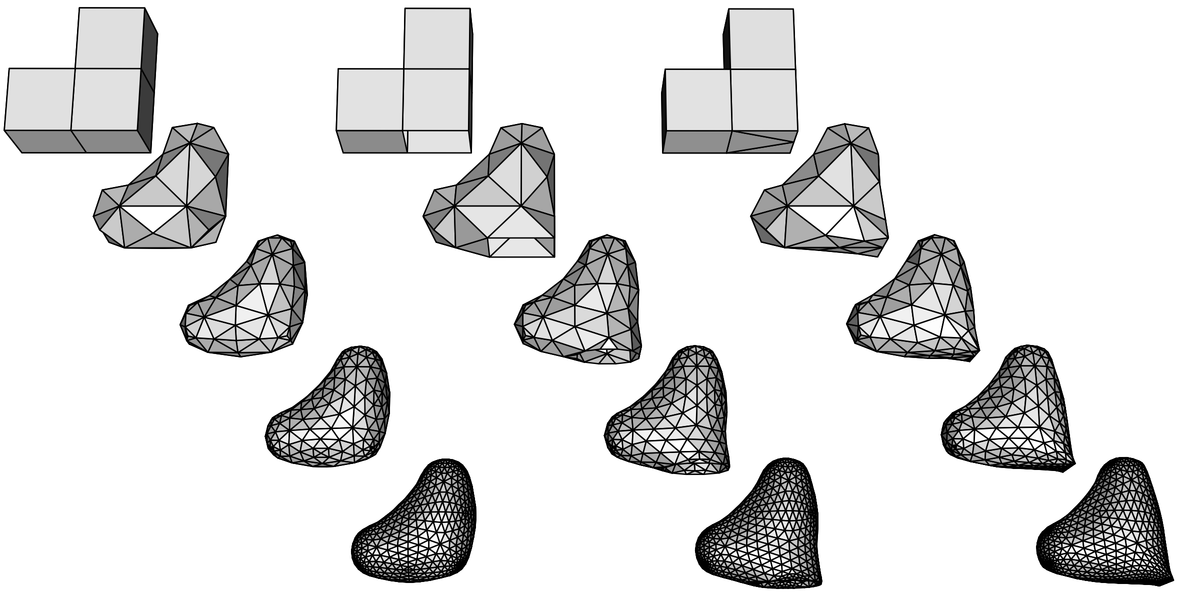
More advanced examples can be found in the Section 26.3 illustrating further the design of the halfedge data structure.

25.6 Advanced Example Programs

25.6.1 Example Creating a Subdivision Surface

This program reads a polyhedral surface from the standard input and writes a refined polyhedral surface to the standard output. Input and output are in the Object File Format, OFF, with the common file extension .off, which is also understood by Geomview [Phi96].

The refinement is a single step of the $\sqrt{3}$-scheme for creating a subdivision surface [Kob00]. Each step subdivides a facet into triangles around a new center vertex, smoothes the position of the old vertices, and flips the old edges. The program is organized along this outline. In each of these parts, the program efficiently uses the knowledge that the newly created vertices, edges, and facets have been added to the end of the sequences. The program needs additional processing memory only for the smoothing step of the old vertices.



The above figure shows three example objects, each subdivided four times. The initial object for the left sequence is the closed surface of three unit cubes glued together to a corner. The example program shown here can handle only closed surfaces, but the extended example `examples/Polyhedron/polyhedron_prog_subdiv_with_boundary.cpp` handles surfaces with boundary. So, the middle sequence starts with the same surface where one of the facets has been removed. The boundary subdivides to a nice circle. The third sequence creates a sharp edge using a trick in the object presentation. The sharp edge is actually a hole whose vertex coordinates pinch the hole shut to form an edge. The example directory `examples/Polyhedron/` contains the OFF files used here.

```
#include <CGAL/Cartesian.h>
#include <CGAL/Polyhedron_3.h>
#include <CGAL/IO/Polyhedron_iostream.h>
#include <iostream>
#include <algorithm>
#include <vector>
#include <cmath>

typedef CGAL::Cartesian<double>                                Kernel;
typedef Kernel::Vector_3                                       Vector;
typedef Kernel::Point_3                                        Point;
typedef CGAL::Polyhedron_3<Kernel>                             Polyhedron;

typedef Polyhedron::Vertex                                     Vertex;
typedef Polyhedron::Vertex_iterator
   Vertex_iterator;
typedef Polyhedron::Halfedge_handle
   Halfedge_handle;
typedef Polyhedron::Edge_iterator
   Edge_iterator;
typedef Polyhedron::Facet_iterator
   Facet_iterator;
```

```
typedef Polyhedron::Halfedge_around_vertex_const_circulator
    HV_circulator;
typedef Polyhedron::Halfedge_around_facet_circulator
    HF_circulator;

void create_center_vertex( Polyhedron& P, Facet_iterator f) {
    Vector vec( 0.0, 0.0, 0.0);
    std::size_t order = 0;
    HF_circulator h = f->facet_begin();
    do {
        vec = vec + ( h->vertex()->point() - CGAL::ORIGIN);
        ++ order;
    } while ( ++h != f->facet_begin());
    CGAL_assertion( order >= 3); // guaranteed by definition of
    polyhedron
    Point center =  CGAL::ORIGIN + (vec / static_cast<double>(order));
    Halfedge_handle new_center = P.create_center_vertex( f->halfedge());
    new_center->vertex()->point() = center;
}

struct Smooth_old_vertex {
    Point operator()( const Vertex& v) const {
        CGAL_precondition((CGAL::circulator_size( v.vertex_begin()) & 1)
    == 0);
        std::size_t degree = CGAL::circulator_size( v.vertex_begin()) /
    2;
        double alpha = ( 4.0 - 2.0 * std::cos( 2.0 * CGAL_PI / degree)) /
     9.0;
        Vector vec = (v.point() - CGAL::ORIGIN) * ( 1.0 - alpha);
        HV_circulator h = v.vertex_begin();
        do {
            vec = vec + ( h->opposite()->vertex()->point() -
    CGAL::ORIGIN)
              * alpha / static_cast<double>(degree);
            ++ h;
            CGAL_assertion( h != v.vertex_begin()); // even degree
    guaranteed
            ++ h;
        } while ( h != v.vertex_begin());
        return (CGAL::ORIGIN + vec);
    }
};

void flip_edge( Polyhedron& P, Halfedge_handle e) {
    Halfedge_handle h = e->next();
    P.join_facet( e);
    P.split_facet( h, h->next()->next());
}

void subdiv( Polyhedron& P) {
    if ( P.size_of_facets() == 0)
        return;
    // We use that new vertices/halfedges/facets are appended at the end.
    std::size_t nv = P.size_of_vertices();
```

```
    Vertex_iterator last_v = P.vertices_end();
    -- last_v;  // the last of the old vertices
    Edge_iterator last_e = P.edges_end();
    -- last_e;  // the last of the old edges
    Facet_iterator last_f = P.facets_end();
    -- last_f;  // the last of the old facets

    Facet_iterator f = P.facets_begin();    // create new center vertices
    do {
        create_center_vertex( P, f);
    } while ( f++ != last_f);

    std::vector<Point> pts;                       // smooth the old vertices
    pts.reserve( nv);  // get intermediate space for the new points
    ++ last_v; // make it the past-the-end position again
    std::transform( P.vertices_begin(), last_v, std::back_inserter( pts),
                    Smooth_old_vertex());
    std::copy( pts.begin(), pts.end(), P.points_begin());

    Edge_iterator e = P.edges_begin();              // flip the old edges
    ++ last_e; // make it the past-the-end position again
    while ( e != last_e) {
        Halfedge_handle h = e;
        ++e; // careful, incr. before flip since flip destroys current
   edge
        flip_edge( P, h);
    };
    CGAL_postcondition( P.is_valid());
}

int main() {
    Polyhedron P;
    std::cin >> P;
    P.normalize_border();
    if ( P.size_of_border_edges() != 0) {
        std::cerr << "The input object has border edges. Cannot
   subdivide."
                  << std::endl;
        std::exit(1);
    }
    subdiv( P);
    std::cout << P;
    return 0;
}
```

File: examples/Polyhedron/polyhedron_prog_subdiv.cpp

25.6.2 Example Using the Incremental Builder and Modifier Mechanism

A utility class *CGAL::Polyhedron_incremental_builder_3* helps in creating polyhedral surfaces from a list of points followed by a list of facets that are represented as indices into the point list. This is particularly useful for implementing file reader for common file formats. It is used here to create a triangle.

A modifier mechanism allows to access the internal representation of the polyhedral surface, i.e., the halfedge data structure, in a controlled manner. A modifier is basically a callback mechanism using a function object. When called, the function object receives the internal halfedge data structure as a parameter and can modify it. On return, the polyhedron can check the halfedge data structure for validity. Such a modifier object must always return with a halfedge data structure that is a valid polyhedral surface. The validity check is implemented as an expensive postcondition at the end of the *delegate()* member function, i.e., it is not called by default, only when expensive checks are activated.

In this example, *Build_triangle* is such a function object derived from *CGAL::Modifier_base<HalfedgeDS>*. The *delegate()* member function of the polyhedron accepts this function object and calls its *operator()* with a reference to its internally used halfedge data structure. Thus, this member function in *Build_triangle* can create the triangle in the halfedge data structure.

```
#include <CGAL/Simple_cartesian.h>
#include <CGAL/Polyhedron_incremental_builder_3.h>
#include <CGAL/Polyhedron_3.h>

// A modifier creating a triangle with the incremental builder.
template <class HDS>
class Build_triangle : public CGAL::Modifier_base<HDS> {
public:
    Build_triangle() {}
    void operator()( HDS& hds) {
        // Postcondition: 'hds' is a valid polyhedral surface.
        CGAL::Polyhedron_incremental_builder_3<HDS> B( hds, true);
        B.begin_surface( 3, 1, 6);
        typedef typename HDS::Vertex   Vertex;
        typedef typename Vertex::Point Point;
        B.add_vertex( Point( 0, 0, 0));
        B.add_vertex( Point( 1, 0, 0));
        B.add_vertex( Point( 0, 1, 0));
        B.begin_facet();
        B.add_vertex_to_facet( 0);
        B.add_vertex_to_facet( 1);
        B.add_vertex_to_facet( 2);
        B.end_facet();
        B.end_surface();
    }
};

typedef CGAL::Simple_cartesian<double>     Kernel;
typedef CGAL::Polyhedron_3<Kernel>         Polyhedron;
typedef Polyhedron::HalfedgeDS             HalfedgeDS;

int main() {
    Polyhedron P;
    Build_triangle<HalfedgeDS> triangle;
    P.delegate( triangle);
    CGAL_assertion( P.is_triangle( P.halfedges_begin()));
    return 0;
}
```

File: examples/Polyhedron/polyhedron_prog_incr_builder.cpp

3D Polyhedral Surfaces Reference Manual

Lutz Kettner

Polyhedral surfaces in three dimensions are composed of vertices, edges, facets and an incidence relationship on them. The organization beneath is a halfedge data structure, which restricts the class of representable surfaces to orientable 2-manifolds – with and without boundary. If the surface is closed we call it a *polyhedron*.

The polyhedral surface is realized as a container class managing vertices, halfedges, facets with their incidences, and maintaining the combinatorial integrity of them. Its local types for the vertices, halfedges and facets are documented separately. A default traits class, a default items class and an incremental builder conclude the references. The polyhedral surface is based on the highly flexible design of the halfedge data structure, see the reference for *HalfedgeDS* in Chapter 26.3.7 or [Ket99], but the default instantiation of the polyhedral surface can be used without knowing the halfedge data structure.

25.7 Classified Reference Pages

Concepts

Classes

Functions

template <class Traits>

ostream& *ostream& out* << *CGAL::Polyhedron_3<Traits> P*

page 1828

template <class Traits>
istream& *istream& in* >> *CGAL::Polyhedron_3<Traits>& P*

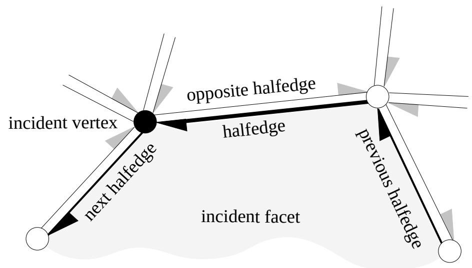
page 1829

25.8 Alphabetical List of Reference Pages

CGAL::Polyhedron_3<Traits>

Definition

A polyhedral surface *Polyhedron_3<Traits>* consists of vertices V, edges E, facets F and an incidence relation on them. Each edge is represented by two halfedges with opposite orientations.



Vertices represent points in 3d-space. Edges are straight line segments between two endpoints. Facets are planar polygons without holes defined by the circular sequence of halfedges along their boundary. The polyhedral surface itself can have holes. The halfedges along the boundary of a hole are called *border halfedges* and have no incident facet. An edge is a *border edge* if one of its halfedges is a border halfedge. A surface is *closed* if it contains no border halfedges. A closed surface is a boundary representation for polyhedra in three dimensions. The convention is that the halfedges are oriented counterclockwise around facets as seen from the outside of the polyhedron. An implication is that the halfedges are oriented clockwise around the vertices. The notion of the solid side of a facet as defined by the halfedge orientation extends to polyhedral surfaces with border edges although they do not define a closed object. If normal vectors are considered for the facets, normals point outwards (following the right hand rule).

The strict definition can be found in [Ket99]. One implication of this definition is that the polyhedral surface is always an orientable and oriented 2-manifold with border edges, i.e., the neighborhood of each point on the polyhedral surface is either homeomorphic to a disc or to a half disc, except for vertices where many holes and surfaces with boundary can join. Another implication is that the smallest representable surface is a triangle (for polyhedral surfaces with border edges) or a tetrahedron (for polyhedra). Boundary representations of orientable 2-manifolds are closed under Euler operations. They are extended with operations that create or close holes in the surface.

Other intersections besides the incidence relation are not allowed, although they are not automatically handled, since self intersections are not easy to check efficiently. *Polyhedron_3<Traits>* does only maintain the combinatorial integrity of the polyhedral surface (using Euler operations) and does not consider the coordinates of the points or any geometric information.

The class *Polyhedron_3<Traits>* can represent polyhedral surfaces as well as polyhedra. The interface is designed in such a way that it is easy to ignore border edges and work only with polyhedra.

The sequence of edges can be ordered in the data structure on request such that the sequence starts with the non-border edges and ends with the border edges. Border edges are then itself ordered such that the halfedge which is incident to the facet comes first and the halfedge incident to the hole comes thereafter. This normalization step counts simultaneously the number of border edges. This number is zero if and only if the surface is a closed polyhedron. Note that this class does not maintain this counter nor the halfedge order during further modifications. There is no automatic caching done for auxiliary information.

#include <CGAL/Polyhedron_3.h>

Parameters

The full template declaration of *Polyhedron_3<Traits>* states four template parameters:

template < class PolyhedronTraits_3,
class PolyhedronItems_3 = CGAL::Polyhedron_items_3,
template < class T, class I> class HalfedgeDS = CGAL::HalfedgeDS_default,
class Alloc = CGAL_ALLOCATOR(int)>
class Polyhedron_3;

The first parameter requires a model of the *PolyhedronTraits_3* concept as argument, for example *CGAL::Polyhedron_traits_3*. The second parameter expects a model of the *PolyhedronItems_3* concept. By default, the class *CGAL::Polyhedron_items_3* is preselected. The third parameter is a class template. A model of the *HalfedgeDS* concept is expected. By default, the class *CGAL::HalfedgeDS_default* is preselected, which is a list based implementation of the halfedge data structure. The fourth parameter *Alloc* requires a standard allocator for STL container classes. The *rebind* mechanism from *Alloc* will be used to create appropriate allocators internally. A default is provided with the macro *CGAL_ALLOCATOR(int)* from the *<CGAL/memory.h>* header file.

Types

Polyhedron_3<Traits>:: Traits	traits class selected for *PolyhedronTraits_3*.
Polyhedron_3<Traits>:: Items	items class selected for *PolyhedronItems_3*.
Polyhedron_3<Traits>:: HalfedgeDS	instantiated halfedge data structure.
Polyhedron_3<Traits>:: size_type	size type of *HalfedgeDS*.
Polyhedron_3<Traits>:: difference_type	difference type of *HalfedgeDS*.
Polyhedron_3<Traits>:: iterator_category	iterator category of *HalfedgeDS* for all iterators.
Polyhedron_3<Traits>:: circulator_category	circulator category of all circulators; bidirectional category if the *Items::Halfedge* provides a *prev()* member function, otherwise forward category.
Polyhedron_3<Traits>:: allocator_type	allocator type *Alloc*.
Polyhedron_3<Traits>:: Vertex	vertex type.
Polyhedron_3<Traits>:: Halfedge	halfedge type.
Polyhedron_3<Traits>:: Facet	facet type.
Polyhedron_3<Traits>:: Point_3	point stored in vertices.
Polyhedron_3<Traits>:: Plane_3	plane equation stored in facets (if supported).

The following handles, iterators, and circulators have appropriate non-mutable counterparts, i.e., *const_handle*, *const_iterator*, and *const_circulator*. The mutable types are assignable to their non-mutable counterparts. Both circulators are assignable to the *Halfedge_iterator*. The iterators are assignable to the respective handle types. Wherever the handles appear in function parameter lists, the corresponding iterators can be used as well. For convenience, the *Edge_iterator* enumerates every other halfedge. It is based on the *CGAL::N_step_adaptor* class. For convenience, the *Point_iterator* enumerates all points in the polyhedral surface in the same order as the *Vertex_iterator*, but with the value type *Point*. It is based on the *CGAL::Iterator_project* adaptor. Similarly, a *Plane_iterator* is provided.

Polyhedron_3<Traits>:: Vertex_handle	handle to vertex.

Polyhedron_3<Traits>:: Halfedge_handle	handle to halfedge.
Polyhedron_3<Traits>:: Facet_handle	handle to facet.

Polyhedron_3<Traits>:: Vertex_iterator	iterator over all vertices.
Polyhedron_3<Traits>:: Halfedge_iterator	iterator over all halfedges.
Polyhedron_3<Traits>:: Facet_iterator	iterator over all facets.

Polyhedron_3<Traits>:: Halfedge_around_vertex_circulator

circulator of halfedges around a vertex (cw).

Polyhedron_3<Traits>:: Halfedge_around_facet_circulator

circulator of halfedges around a facet (ccw).

Polyhedron_3<Traits>:: Edge_iterator	iterator over all edges (every other halfedge).
Polyhedron_3<Traits>:: Point_iterator	iterator over all points.
Polyhedron_3<Traits>:: Plane_iterator	iterator over all plane equations.

advanced

Types for Tagging Optional Features

The following types are equal to either *CGAL::Tag_true* or *CGAL::Tag_false*, depending on whether the named feature is supported or not.

Polyhedron_3<Traits>:: Supports_vertex_halfedge	*Vertex::halfedge().*
Polyhedron_3<Traits>:: Supports_vertex_point	*Vertex::point().*
Polyhedron_3<Traits>:: Supports_halfedge_prev	*Halfedge::prev().*
Polyhedron_3<Traits>:: Supports_halfedge_vertex	*Halfedge::vertex().*
Polyhedron_3<Traits>:: Supports_halfedge_facet	*Halfedge::facet().*
Polyhedron_3<Traits>:: Supports_facet_halfedge	*Facet::halfedge().*
Polyhedron_3<Traits>:: Supports_facet_plane	*Facet::plane().*
Polyhedron_3<Traits>:: Supports_removal	supports removal of individual elements.

advanced

Creation

Polyhedron_3<Traits> P(Traits traits = Traits());

Polyhedron_3<Traits> P(size_type v, size_type h, size_type f, Traits traits = Traits());

a polyhedron *P* with storage reserved for *v* vertices, *h* halfedges, and *f* facets. The reservation sizes are a hint for optimizing storage allocation.

void *P.reserve(size_type v, size_type h, size_type f)*

reserve storage for *v* vertices, *h* halfedges, and *f* facets. The reservation sizes are a hint for optimizing storage allocation. If the *capacity* is already greater than the requested size nothing happens. If the *capacity* changes all iterators and circulators might invalidate.

Halfedge_handle	*P.make_tetrahedron()*	

a tetrahedron is added to the polyhedral surface. Returns a halfedge of the tetrahedron.

Halfedge_handle	*P.make_tetrahedron(Point p1, Point p2, Point p3, Point p4)*	

a tetrahedron is added to the polyhedral surface with its vertices initialized to p_1, p_2, p_3, and p_4. Returns that halfedge of the tetrahedron which incident vertex is initialized to p_1. The incident vertex of the next halfedge is p_2, and the vertex thereafter is p_3. The remaining fourth vertex is initialized to p_4.

Halfedge_handle	*P.make_triangle()*	

a triangle with border edges is added to the polyhedral surface. Returns a non-border halfedge of the triangle.

Halfedge_handle	*P.make_triangle(Point p1, Point p2, Point p3)*	

a triangle with border edges is added to the polyhedral surface with its vertices initialized to p_1, p_2, and p_3. Returns that non-border halfedge of the triangle which incident vertex is initialized to p_1. The incident vertex of the next halfedge is p_2, and the vertex thereafter is p_3.

Access Member Functions

bool	*P.empty() const*	returns true if *P* is empty.
size_type	*P.size_of_vertices() const*	number of vertices.
size_type	*P.size_of_halfedges() const*	number of halfedges (incl. border halfedges).
size_type	*P.size_of_facets() const*	number of facets.
size_type	*P.capacity_of_vertices() const*	space reserved for vertices.
size_type	*P.capacity_of_halfedges() const*	space reserved for halfedges.
size_type	*P.capacity_of_facets() const*	space reserved for facets.
size_t	*P.bytes() const*	bytes used for the polyhedron.
size_t	*P.bytes_reserved() const*	bytes reserved for the polyhedron.
allocator_type	*P.get_allocator() const*	allocator object.
Vertex_iterator	*P.vertices_begin()*	iterator over all vertices.
Vertex_iterator	*P.vertices_end()*	past-the-end iterator.
Halfedge_iterator	*P.halfedges_begin()*	iterator over all halfedges.
Halfedge_iterator	*P.halfedges_end()*	past-the-end iterator.
Facet_iterator	*P.facets_begin()*	iterator over all facets (excluding holes).

Facet_iterator	*P.facets_end()*	past-the-end iterator.
Edge_iterator	*P.edges_begin()*	iterator over all edges.
Edge_iterator	*P.edges_end()*	past-the-end iterator.
Point_iterator	*P.points_begin()*	iterator over all points.
Point_iterator	*P.points_end()*	past-the-end iterator.
Plane_iterator	*P.planes_begin()*	iterator over all plane equations.
Plane_iterator	*P.planes_end()*	past-the-end iterator.
Traits	*P.traits() const*	returns the traits class.

Combinatorial Predicates

bool *P.is_closed() const*

returns *true* if there are no border edges.

bool *P.is_pure_bivalent() const*

returns *true* if all vertices have exactly two incident edges.

bool *P.is_pure_trivalent() const*

returns *true* if all vertices have exactly three incident edges.

bool *P.is_pure_triangle() const*

returns *true* if all facets are triangles.

bool *P.is_pure_quad() const*

returns *true* if all facets are quadrilaterals.

bool *P.is_triangle(Halfedge_const_handle h) const*

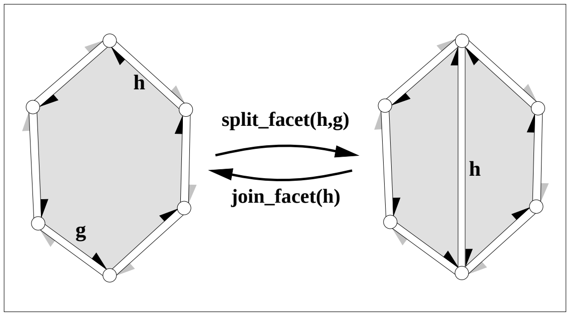
true iff the connected component denoted by *h* is a triangle.

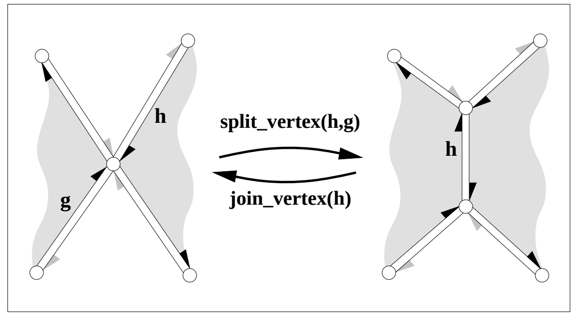
bool *P.is_tetrahedron(Halfedge_const_handle h) const*

true iff the connected component denoted by *h* is a tetrahedron.

Euler Operators (Combinatorial Modifications)

The following Euler operations modify consistently the combinatorial structure of the polyhedral surface. The geometry remains unchanged.





Halfedge_handle *P.split_facet(Halfedge_handle h, Halfedge_handle g)*

splits the facet incident to *h* and *g* into two facets with a new diagonal between the two vertices denoted by *h* and *g* respectively. The second (new) facet is a copy of the first facet. Returns *h->next()* after the operation, i.e., the new diagonal. The new face is to the right of the new diagonal, the old face is to the left. The time is proportional to the distance from *h* to *g* around the facet.
Precondition: *h* and *g* are incident to the same facet. *h != g* (no loops). *h->next() != g* and *g->next() != h* (no multi-edges).

Halfedge_handle *P.join_facet(Halfedge_handle h)*

joins the two facets incident to *h*. The facet incident to *h->opposite()* gets removed. Both facets might be holes. Returns the predecessor of *h* around the facet. The invariant *join_facet(split_facet(h, g))* returns *h* and keeps the polyhedron unchanged. The time is proportional to the size of the facet removed and the time to compute *h->prev()*.
Precondition: The degree of both vertices incident to *h* is at least three (no antennas).
Requirement: *Supports_removal* $\equiv$ *CGAL::Tag_true*.

Halfedge_handle *P.split_vertex(Halfedge_handle h, Halfedge_handle g)*

splits the vertex incident to *h* and *g* into two vertices, the old vertex remains and a new copy is created, and connects them with a new edge. Let *hnew* be *h->next()->opposite()* after the split, i.e., a halfedge of the new edge. The split regroups the halfedges around the two vertices. The halfedge sequence *hnew*, *g->next()->opposite()*, ..., *h* remains around the old vertex, while the halfedge sequence *hnew->opposite()*, *h->next()->opposite()* (before the split), ..., *g* is regrouped around the new vertex. The split returns *hnew*, i.e., the new halfedge incident to the old vertex. The time is proportional to the distance from *h* to *g* around the vertex.
Precondition: *h* and *g* are incident to the same vertex. *h != g* (antennas are not allowed).
Note: A special application of the split is *split_vertex(h,h->next()->opposite())* which is equivalent to an edge split of the halfedge *h->next()* that creates a new vertex on the halfedge *h->next()*. See also *split_edge(h)* below.

Halfedge_handle *P.join_vertex(Halfedge_handle h)*

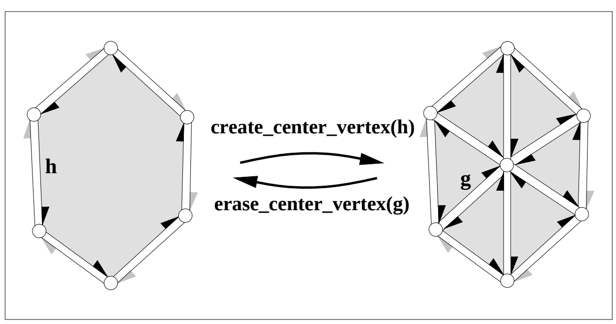
joins the two vertices incident to *h*. The vertex denoted by *h->opposite()* gets removed. Returns the predecessor of *h* around the vertex, i.e., *h->opposite()->prev()*. The invariant *join_vertex(split_vertex(h, g))* returns *h* and keeps the polyhedron unchanged. The time is proportional to the degree of the vertex removed and the time to compute *h->prev()* and *h->opposite()->prev()*.
Precondition: The size of both facets incident to *h* is at least four (no multi-edges).
Requirement: *Supports_removal* ≡ *CGAL::Tag_true*.

Halfedge_handle *P.split_edge(Halfedge_handle h)*

splits the halfedge *h* into two halfedges inserting a new vertex that is a copy of *h->opposite()->vertex()*. Is equivalent to *split_vertex(h->prev(), h->opposite())*. The call of *prev()* can make this method slower than a direct call of *split_vertex()* if the previous halfedge is already known and computing it would be costly when the halfedge data structure does not support the *prev()* member function. Returns the new halfedge *hnew* pointing to the inserted vertex. The new halfedge is followed by the old halfedge, i.e., *hnew->next() == h*.

Halfedge_handle *P.flip_edge(Halfedge_handle h)*

performs an edge flip. It returns *h* after rotating the edge *h* one vertex in the direction of the face orientation.
Precondition: *h != Halfedge_handle()* and both facets incident to *h* are triangles.



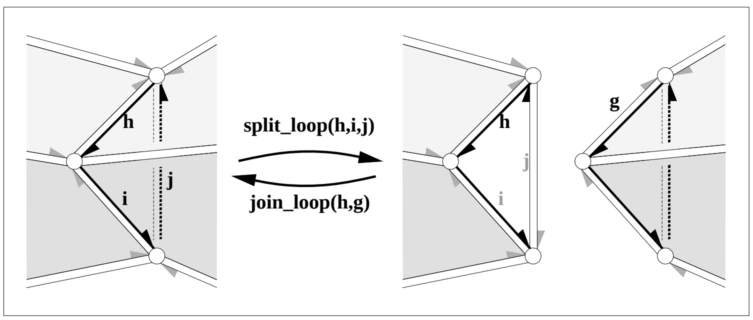
Halfedge_handle *P.create_center_vertex(Halfedge_handle h)*

barycentric triangulation of *h->facet()*. Creates a new vertex, a copy of *h->vertex()*, and connects it to each vertex incident to *h->facet()* splitting *h->facet()* into triangles. *h* remains incident to the original facet, all other triangles are copies of this facet. Returns the halfedge *h->next()* after the operation, i.e., a halfedge pointing to the new vertex. The time is proportional to the size of the facet.
Precondition: *h* is not a border halfedge.

Halfedge_handle *P.erase_center_vertex(Halfedge_handle g)*

reverses *create_center_vertex*. Erases the vertex pointed to by g and all incident halfedges thereby merging all incident facets. Only *g->facet()* remains. The neighborhood of *g->vertex()* may not be triangulated, it can have larger facets. Returns the halfedge *g->prev()*. Thus, the invariant h == *erase_center_vertex(create_center_vertex(h))* holds if h is not a border halfedge. The time is proportional to the sum of the size of all incident facets.
Precondition: None of the incident facets of *g->vertex()* is a hole. There are at least two distinct facets incident to the facets that are incident to *g->vertex()*. (This prevents the operation from collapsing a volume into two facets glued together with opposite orientations, such as would happen with any vertex of a tetrahedron.)
Requirement: *Supports_removal* $\equiv$ *CGAL::Tag_true*.

Euler Operators Modifying Genus



Halfedge_handle *P.split_loop(Halfedge_handle h, Halfedge_handle i, Halfedge_handle j)*

cuts the polyhedron into two parts along the cycle (h,i,j) (edge j runs on the backside of the three dimensional figure above). Three new vertices (one copy for each vertex in the cycle) and three new halfedges (one copy for each halfedge in the cycle), and two new triangles are created. h,i,j will be incident to the first new triangle. The return value will be the halfedge incident to the second new triangle which is the copy of *h-opposite()*.
Precondition: h,i,j denote distinct, consecutive vertices of the polyhedron and form a cycle: i.e., *h->vertex()* == *i->opposite()->vertex()*, ..., *j->vertex()* == *h->opposite()->vertex()*. The six facets incident to h,i,j are all distinct.

Halfedge_handle *P.join_loop(Halfedge_handle h, Halfedge_handle g)*

glues the boundary of the two facets denoted by h and g together and returns h. Both facets and the vertices along the facet denoted by g gets removed. Both facets may be holes. The invariant *join_loop(h, split_loop(h, i, j))* returns h and keeps the polyhedron unchanged.
Precondition: The facets denoted by h and g are different and have equal degree (i.e., number of edges).
Requirement: *Supports_removal* $\equiv$ *CGAL::Tag_true*.

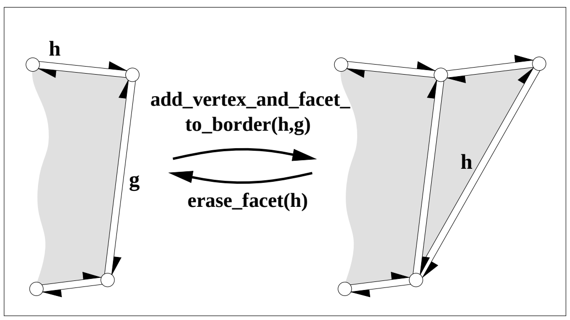
Modifying Facets and Holes

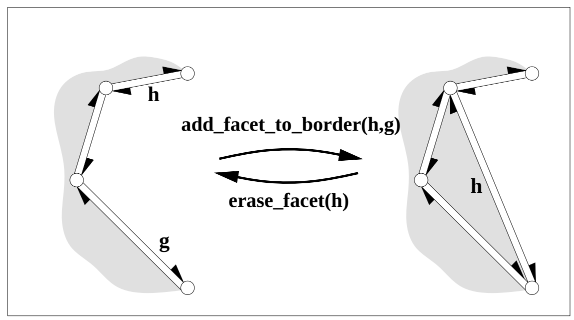
Halfedge_handle *P.make_hole(Halfedge_handle h)*

removes the incident facet of *h* and changes all halfedges incident to the facet into border edges. Returns *h*. See *erase_facet(h)* for a more generalized variant.
Precondition: None of the incident halfedges of the facet is a border edge.
Requirement: *Supports_removal* ≡ *CGAL::Tag_true*.

Halfedge_handle *P.fill_hole(Halfedge_handle h)*

fills a hole with a newly created facet. Makes all border halfedges of the hole denoted by *h* incident to the new facet. Returns *h*.
Precondition: *h.is_border()*.





Halfedge_handle *P.add_vertex_and_facet_to_border(Halfedge_handle h, Halfedge_handle g)*

creates a new facet within the hole incident to *h* and *g* by connecting the tip of *g* with the tip of *h* with two new halfedges and a new vertex and filling this separated part of the hole with a new facet, such that the new facet is incident to *g*. Returns the halfedge of the new edge that is incident to the new facet and the new vertex.
Precondition: *h->is_border()*, *g->is_border()*, *h != g*, and *g* can be reached along the same hole starting with *h*.

Halfedge_handle *P.add_facet_to_border(Halfedge_handle h, Halfedge_handle g)*

creates a new facet within the hole incident to *h* and *g* by connecting the vertex denoted by *g* with the vertex denoted by *h* with a new halfedge and filling this separated part of the hole with a new facet, such that the new facet is incident to *g*. Returns the halfedge of the new edge that is incident to the new facet.
Precondition: *h->is_border()*, *g->is_border()*, *h != g*, *h->next() != g*, and *g* can be reached along the same hole starting with *h*.

Erasing

void *P.erase_facet(Halfedge_handle h)*

removes the incident facet of *h* and changes all halfedges incident to the facet into border edges or removes them from the polyhedral surface if they were already border edges. If this creates isolated vertices they get removed as well. See *make_hole(h)* for a more specialized variant.
Precondition: *h->is_border() == false*.
Requirement: *Supports_removal* ≡ *CGAL::Tag_true*.

void *P.erase_connected_component(Halfedge_handle h)*

removes the vertices, halfedges, and facets that belong to the connected component of *h*.
Requirement: *Supports_removal* ≡ *CGAL::Tag_true*.

unsigned int *P.keep_largest_connected_components(unsigned int nb_components_to_keep)*

Erases the small connected components and the isolated vertices. Keep *nb_components_to_keep* largest connected components. Returns the number of connected components erased (ignoring isolated vertices).
Requirement: supports vertices, halfedges, and removal operation.

void *P.clear()*

removes all vertices, halfedges, and facets.

Operations with Border Halfedges

advanced

Halfedges incident to a hole are called *border halfedges*. An halfedge is a *border edge* if itself or its opposite halfedge are border halfedges. The only requirement to work with border halfedges is that the *Halfedge* class provides a member function *is_border()* returning a *bool*. Usually, the halfedge data structure supports facets and a *NULL* facet pointer will indicate a border halfedge, but this is not the only possibility. The *is_border()* predicate divides the edges into two classes, the border edges and the non-border edges. The following normalization reorganizes the sequential storage of the edges such that the non-border edges precede the border edges, and that for each border edge the latter one of the two halfedges is a border halfedge (the first one is a non-border halfedge in conformance with the polyhedral surface definition). The normalization stores the number of border halfedges and the halfedge iterator the border edges start at within the data structure. Halfedge insertion or removal and changing the border status of a halfedge invalidate these values. They are not automatically updated.

void *P.normalize_border()*

sorts halfedges such that the non-border edges precede the border edges. For each border edge the halfedge iterator will reference the halfedge incident to the facet right before the halfedge incident to the hole.

size_type *P.size_of_border_halfedges() const*

number of border halfedges.
Precondition: last *normalize_border()* call still valid, see above.

size_type *P.size_of_border_edges() const*

number of border edges. Since each border edge of a polyhedral surface has exactly one border halfedge, this number is equal to *size_of_border_halfedges()*.
Precondition: last *normalize_border()* call still valid, see above.

Halfedge_iterator *P.border_halfedges_begin()*

halfedge iterator starting with the border edges. The range [*halfedges_begin(), border_halfedges_begin()*) denotes all non-border halfedges. The range [*border_halfedges_begin(), halfedges_end()*) denotes all border edges.
Precondition: last *normalize_border()* call still valid, see above.

Edge_iterator *P.border_edges_begin()*

edge iterator starting with the border edges. The range [*edges_begin(), border_edges_begin()*) denotes all non-border edges. The range [*border_edges_begin(), edges_end()*) denotes all border edges.
Precondition: last *normalize_border()* call still valid, see above.

advanced

Miscellaneous

void *P.inside_out()*

reverses facet orientations (incl. plane equations if supported).

bool *P.is_valid(bool verbose = false, int level = 0) const*

returns *true* if the polyhedral surface is combinatorially consistent. If *verbose* is *true*, statistics are printed to *cerr*. For *level == 1* the normalization of the border edges is checked too. This method checks in particular level 3 of *CGAL::Halfedge_data_structure_decorator::is_valid* from page 26.4 and that each facet is at least a triangle and that the two incident facets of a non-border edge are distinct.

bool *P.normalized_border_is_valid(bool verbose = false) const*

returns *true* if the border halfedges are in normalized representation, which is when enumerating all halfedges with the iterator: The non-border edges precede the border edges and for border edges, the second halfedge is the border halfedge. The halfedge iterator *border_halfedges_begin()* denotes the first border edge. If *verbose* is *true*, statistics are printed to *cerr*.

advanced

void *P.delegate(CGAL::Modifier_base<HDS>& m)*

calls the *operator()* of the modifier *m*. See *CGAL::Modifier_base* for a description of modifier design and its usage.
Precondition: The polyhedral surface must be valid when the modifier returns from execution.

advanced

See Also

CGAL::Polyhedron_3<Traits>::Vertex . . . page 1812
CGAL::Polyhedron_3<Traits>::Halfedge . . . page 1809
CGAL::Polyhedron_3<Traits>::Facet . . . page 1807
PolyhedronTraits_3 . . . page 1823
CGAL::Polyhedron_traits_3<Kernel> . . . page 1824
PolyhedronItems_3 . . . page 1818
CGAL::Polyhedron_items_3 . . . page 1820
HalfedgeDS . . . page 1843
CGAL::HalfedgeDS_default . . . page 1867
CGAL::Polyhedron_incremental_builder_3<HDS> . . . page 1814

CGAL::Modifier_base.

Example

This example program instantiates a polyhedron using the default traits class and creates a tetrahedron.

```
#include <CGAL/Simple_cartesian.h>
#include <CGAL/Polyhedron_3.h>

typedef CGAL::Simple_cartesian<double>     Kernel;
typedef CGAL::Polyhedron_3<Kernel>         Polyhedron;
typedef Polyhedron::Halfedge_handle        Halfedge_handle;

int main() {
    Polyhedron P;
    Halfedge_handle h = P.make_tetrahedron();
    if ( P.is_tetrahedron(h))
        return 0;
    return 1;
}
```

File: examples/Polyhedron/polyhedron_prog_simple.cpp

CGAL::Polyhedron_3<Traits>::Facet

Class

Definition

A facet optionally stores a plane equation, and a reference to an incident halfedge that points to the facet. Type tags indicate whether these member functions are supported. Figure 25.1 depicts the relationship between a halfedge and its incident halfedges, vertices, and facets. The circulator is assignable to the *Halfedge_handle*. The circulator is bidirectional if the halfedge provided to the polyhedron with the *Items* template argument provides a member function *prev()*, otherwise it is of the forward category.

#include <CGAL/Polyhedron_3.h>

Types

Facet:: Vertex	type of incident vertices.
Facet:: Halfedge	type of incident halfedges.
Facet:: Plane_3	plane equation type stored in facets.

Facet:: Vertex_handle	handle to vertex.
Facet:: Halfedge_handle	handle to halfedge.
Facet:: Facet_handle	handle to facet.
Facet:: Halfedge_around_facet_circulator	circulator of halfedges around a facet.

Facet:: Vertex_const_handle
Facet:: Halfedge_const_handle
Facet:: Facet_const_handle
Facet:: Halfedge_around_facet_const_circulator

Facet:: Supports_facet_halfedge	≡ *CGAL::Tag_true* or *CGAL::Tag_false*.
Facet:: Supports_facet_plane	≡ *CGAL::Tag_true* or *CGAL::Tag_false*.

Creation

Facet f;	default constructor.

Operations available if *Supports_facet_plane* ≡ *CGAL::Tag_true*

Plane_3&	*f.plane()*	
const Plane_3&	*f.plane() const*	plane equation.

Operations available if *Supports_facet_halfedge* ≡ *CGAL::Tag_true*

Halfedge_handle	*f.halfedge()*	
Halfedge_const_handle	*f.halfedge() const*	an incident halfedge that points to *f*.

Halfedge_around_facet_circulator		
	f.facet_begin()	
Halfedge_around_facet_const_circulator		
	f.facet_begin() const	
		circulator of halfedges around the facet (counterclockwise).
void	*f.set_halfedge(Halfedge_handle h)*	
		sets incident halfedge to h. *Precondition*: h is incident, i.e., *h->facet()* $==f$.
std::size_t	*f.facet_degree() const*	
		the degree of the facet, i.e., number of edges on the boundary of this facet.
bool	*f.is_triangle() const*	
		returns *true* if the facet is a triangle.
bool	*f.is_quad() const*	returns *true* if the facet is a quadrilateral.

See Also

CGAL::Polyhedron_3<Traits>::Vertex . . . . page 1812
CGAL::Polyhedron_3<Traits>::Halfedge . . . . page 1809
CGAL::Polyhedron_3<Traits> . . . . page 1795

CGAL::Polyhedron_3<Traits>::Halfedge

Definition

Figure 25.1 depicts the relationship between a halfedge and its incident halfedges, vertices, and facets. A halfedge is an oriented edge between two vertices. It is always paired with a halfedge pointing in the opposite direction. The *opposite()* member function returns this halfedge of opposite orientation. If a halfedge is incident to a facet the *next()* member function points to the successor halfedge around this facet. For border edges the *next()* member function points to the successor halfedge along the hole. For more than two border edges at a vertex, the next halfedge along a hole is not uniquely defined, but a consistent assignment of the next halfedge will be maintained in the data structure. An invariant is that successive assignments of the form *h = h->next()* cycle counterclockwise around the facet (or hole) and traverse all halfedges incident to this facet (or hole). A similar invariant is that successive assignments of the form *h = h->next()->opposite()* cycle clockwise around the vertex and traverse all halfedges incident to this vertex. Two circulators are provided for these circular orders.

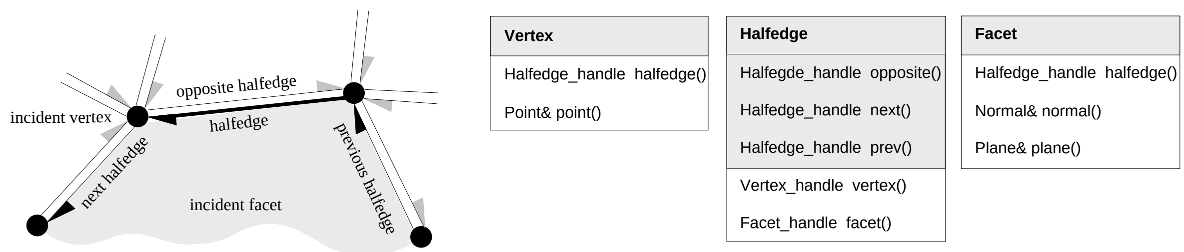


Figure 25.1: The three classes *Vertex*, *Halfedge*, and *Facet* of the polyhedral surface. Member functions with shaded background are mandatory. The others are optionally supported.

The incidences encoded in *opposite()* and *next()* are available for each instantiation of polyhedral surfaces. The other incidences are optionally available as indicated with type tags. The *prev()* member function points to the preceding halfedge around the same facet. It is always available, though it might perform a search around the facet using the *next()* member function to find the previous halfedge if the underlying halfedge data structure does not provide an efficient *prev()* member function for halfedges. Handles to the incident vertex and facet are optionally stored.

The circulators are assignable to the *Halfedge_handle*. The circulators are bidirectional if the halfedge provided to the polyhedron with the *Items* template argument provides a member function *prev()*, otherwise they are of the forward category.

#include <CGAL/Polyhedron_3.h>

Types

Halfedge:: Vertex	type of incident vertices.
Halfedge:: Facet	type of incident facets.
Halfedge:: Vertex_handle	handle to vertex.
Halfedge:: Halfedge_handle	handle to halfedge.
Halfedge:: Facet_handle	handle to facet.
Halfedge:: Halfedge_around_vertex_circulator	circulator of halfedges around a vertex.
Halfedge:: Halfedge_around_facet_circulator	circulator of halfedges around a facet.

Halfedge:: Vertex_const_handle
Halfedge:: Halfedge_const_handle
Halfedge:: Facet_const_handle
Halfedge:: Halfedge_around_vertex_const_circulator
Halfedge:: Halfedge_around_facet_const_circulator

Halfedge:: Supports_halfedge_prev	≡ *CGAL::Tag_true* or *CGAL::Tag_false*.
Halfedge:: Supports_halfedge_vertex	≡ *CGAL::Tag_true* or *CGAL::Tag_false*.
Halfedge:: Supports_halfedge_face	≡ *CGAL::Tag_true* or *CGAL::Tag_false*.

Creation

Halfedge h;	default constructor.

Operations

Halfedge_handle	*h.opposite()*	
Halfedge_const_handle	*h.opposite() const*	the opposite halfedge.
Halfedge_handle	*h.next()*	
Halfedge_const_handle	*h.next() const*	the next halfedge around the facet.
Halfedge_handle	*h.prev()*	
Halfedge_const_handle	*h.prev() const*	the previous halfedge around the facet.
Halfedge_handle	*h.next_on_vertex()*	
Halfedge_const_handle	*h.next_on_vertex() const*	the next halfedge around the vertex (clockwise). Is equal to *h.next()->opposite()*.
Halfedge_handle	*h.prev_on_vertex()*	
Halfedge_const_handle	*h.prev_on_vertex() const*	the previous halfedge around the vertex (counterclockwise). Is equal to *h.opposite()->prev()*.
bool	*h.is_border() const*	is true if *h* is a border halfedge.
bool	*h.is_border_edge() const*	is true if *h* or *h.opposite()* is a border halfedge.

Halfedge_around_vertex_circulator

h.vertex_begin()

Halfedge_around_vertex_const_circulator

h.vertex_begin() const — circulator of halfedges around the vertex (clockwise).

Halfedge_around_facet_circulator

h.facet_begin()

Halfedge_around_facet_const_circulator		
	h.facet_begin() const	circulator of halfedges around the facet (counterclockwise).
std::size_t	*h.vertex_degree() const*	the degree of the incident vertex, i.e., number of edges emanating from this vertex.
bool	*h.is_bivalent() const*	returns *true* if the incident vertex has exactly two incident edges.
bool	*h.is_trivalent() const*	returns *true* if the incident vertex has exactly three incident edges.
std::size_t	*h.facet_degree() const*	the degree of the incident facet, i.e., number of edges on the boundary of this facet.
bool	*h.is_triangle() const*	returns *true* if the incident facet is a triangle.
bool	*h.is_quad() const*	returns *true* if the incident facet is a quadrilateral.

Operations available if *Supports_halfedge_vertex* $\equiv$ *CGAL::Tag_true*

Vertex_handle	*h.vertex()*	
Vertex_const_handle	*h.vertex() const*	the incident vertex of *h*.

Operations available if *Supports_halfedge_facet* $\equiv$ *CGAL::Tag_true*

Facet_handle	*h.facet()*	
Facet_const_handle	*h.facet() const*	the incident facet of *h*. If *h* is a border halfedge the result is default construction of the handle.

See Also

CGAL::Polyhedron_3<Traits>::Vertex . . . . page 1812
CGAL::Polyhedron_3<Traits>::Facet . . . . page 1807
CGAL::Polyhedron_3<Traits> . . . . page 1795

Implementation

The member functions *prev()* and *prev_on_vertex()* work in constant time if *Supports_halfedge_prev* $\equiv$ *CGAL::Tag_true*. Otherwise both methods search for the previous halfedge around the incident facet.

Class

CGAL::Polyhedron_3<Traits>::Vertex

Definition

A vertex optionally stores a point and a reference to an incident halfedge that points to the vertex. Type tags indicate whether these member functions are supported. Figure 25.1 depicts the relationship between a halfedge and its incident halfedges, vertices, and facets. The circulator is assignable to the *Halfedge_handle*. The circulator is bidirectional if the halfedge provided to the polyhedron with the *Items* template argument provides a member function *prev()*, otherwise it is of the forward category.

#include <CGAL/Polyhedron_3.h>

Types

Vertex:: Halfedge	type of incident halfedges.
Vertex:: Facet	type of incident facets.
Vertex:: Point_3	point type stored in vertices.
Vertex:: Vertex_handle	handle to vertex.
Vertex:: Halfedge_handle	handle to halfedge.
Vertex:: Facet_handle	handle to facet.
Vertex:: Halfedge_around_vertex_circulator	circulator of halfedges around a vertex.
Vertex:: Vertex_const_handle	
Vertex:: Halfedge_const_handle	
Vertex:: Facet_const_handle	
Vertex:: Halfedge_around_vertex_const_circulator	
Vertex:: Supports_vertex_halfedge	≡ *CGAL::Tag_true* or *CGAL::Tag_false*.
Vertex:: Supports_vertex_point	≡ *CGAL::Tag_true* or *CGAL::Tag_false*.

Creation

Vertex v;	default constructor.
Vertex v(Point p);	vertex initialized with a point.

Operations available if *Supports_vertex_point* ≡ *CGAL::Tag_true*

Point_3&	*v.point()*	
const Point_3&	*v.point() const*	the point.

Operations available if *Supports_vertex_halfedge* ≡ *CGAL::Tag_true*

Halfedge_handle	*v.halfedge()*	
Halfedge_const_handle	*v.halfedge() const*	an incident halfedge that points to *v*.
Halfedge_around_vertex_circulator		
	v.vertex_begin()	
Halfedge_around_vertex_const_circulator		
	v.vertex_begin() const	
		circulator of halfedges around the vertex (clockwise).
void	*v.set_halfedge(Halfedge_handle h)*	
		sets incident halfedge to *h*. *Precondition*: *h* is incident, i.e., *h->vertex() == v*.
std::size_t	*v.vertex_degree() const*	
		the degree of the vertex, i.e., number of edges emanating from this vertex.
bool	*v.is_bivalent() const*	
		returns *true* if the vertex has exactly two incident edges.
bool	*v.is_trivalent() const*	
		returns *true* if the vertex has exactly three incident edges.

See Also

CGAL::Polyhedron_3<Traits>::Halfedge . . . . page 1809
CGAL::Polyhedron_3<Traits>::Facet . . . . page 1807
CGAL::Polyhedron_3<Traits> . . . . page 1795

Class

CGAL::Polyhedron_incremental_builder_3<HDS>

Definition

The auxiliary class *Polyhedron_incremental_builder_3<HDS>* supports the incremental construction of polyhedral surfaces, which is for example convenient when constructing polyhedral surfaces from file formats, such as the Object File Format (OFF) [Phi96], OpenInventor [Wer94] or VRML [BPP95, VRM96]. *Polyhedron_incremental_builder_3<HDS>* needs access to the internal halfedge data structure of type *HDS* of the polyhedral surface. It is intended to be used within a modifier, see *CGAL::Modifier_base*.

The incremental builder might be of broader interest for other uses of the halfedge data structures, but it is specifically bound to the definition of polyhedral surfaces given here. During construction all conditions of polyhedral surfaces are checked and in case of violation an error status is set. A diagnostic message will be issued to *cerr* if the *verbose* flag has been set at construction time.

The incremental construction starts with a list of all point coordinates and concludes with a list of all facet polygons. Edges are not explicitly specified. They are derived from the vertex incidence information provided from the facet polygons. The polygons are given as a sequence of vertex indices. The halfedge data structure *HDS* must support vertices (i.e., *Supports_halfedge_vertex* $\equiv$ *CGAL::Tag_true*). Vertices and facets can be added in arbitrary order as long as a call to *add_vertex_to_facet()* refers only to a vertex index that is already known. Some methods return already handles to vertices, facets, and halfedges newly constructed. They can be used to initialize additional fields, however, the incidences in the halfedge-data structure are not stable and are not allowed to be changed.

The incremental builder can work in two modes: *RELATIVE_INDEXING* (the default), in which a polyhedral surface already contained in the halfedge data structure is ignored and all indices are relative to the newly added surface, or *ABSOLUTE_INDEXING*, in which all indices are absolute indices including an already existing polyhedral surface. The former mode allows to create easily independent connected components, while the latter mode allows to to continue the construction of an existing surface, the absolute indexing allows to address existing vertices when creating new facets.

#include <CGAL/Polyhedron_incremental_builder_3.h>

Types

Polyhedron_incremental_builder_3<HDS>:: HalfedgeDS	halfedge data structure *HDS*.
Polyhedron_incremental_builder_3<HDS>:: Point_3	point type of the vertex.
Polyhedron_incremental_builder_3<HDS>:: size_type	size type.

typedef typename HalfedgeDS::Vertex_handle	*Vertex_handle;*
typedef typename HalfedgeDS::Halfedge_handle	*Halfedge_handle;*
typedef typename HalfedgeDS::Face_handle	*Facet_handle;*

Constants

enum { *RELATIVE_INDEXING, ABSOLUTE_INDEXING*}*;*

two different indexing modes.

Creation

Polyhedron_incremental_builder_3<HDS> B(HDS& hds, bool verbose = false);

stores a reference to the halfedge data structure *hds* of a polyhedral surface in its internal state. An existing polyhedral surface in *hds* remains unchanged. The incremental builder appends the new polyhedral surface. If *verbose* is *true*, diagnostic messages will be printed to *cerr* in case of malformed input data.

Surface Creation

To build a polyhedral surface, the following regular expression gives the correct and allowed order and nesting of method calls from this section:

$$begin\_surface\ (add\_vertex \mid (begin\_facet\ add\_vertex\_to\_facet * end\_facet)) * end\_surface$$

void *B.begin_surface(size_type v, size_type f, size_type h = 0, int mode = RELATIVE_INDEXING)*

starts the construction. v is the number of new vertices to expect, f the number of new facets, and h the number of new halfedges. If h is unspecified (== *0*) it is estimated using Euler's equation (plus 5% for the so far unknown holes and genus of the object). These values are used to reserve space in the halfedge data structure *hds*. If the representation supports insertion these values do not restrict the class of constructible polyhedra. If the representation does not support insertion the object must fit into the reserved sizes.
If *mode* is set to *ABSOLUTE_INDEXING* the incremental builder uses absolute indexing and the vertices of the old polyhedral surface can be used in new facets (needs preprocessing time linear in the size of the old surface). Otherwise relative indexing is used starting with new indices for the new construction.

Vertex_handle *B.add_vertex(Point_3 p)*

adds a new vertex for p and returns its handle.

Facet_handle *B.begin_facet()* starts a new facet and returns its handle.

void *B.add_vertex_to_facet(size_type i)*

adds a vertex with index i to the current facet. The first point added with *add_vertex()* has the index 0 if *mode* was set to *RELATIVE_INDEXING*, otherwise the first vertex in the referenced *hds* has the index 0.

Halfedge_handle *B.end_facet()* ends a newly constructed facet. Returns the handle to the halfedge incident to the new facet that points to the vertex added first. The halfedge can be safely used to traverse the halfedge cycle around the new facet.

void *B.end_surface()* ends the construction.

Additional Operations

template <class InputIterator>
Halfedge_handle *B.add_facet(InputIterator first, InputIterator beyond)*

is a synonym for *begin_facet()*, a call to *add_vertex_to_facet()* for each value in the range *[first,beyond)*, and a call to *end_facet()*. Returns the return value of *end_facet()*.
Precondition: The value type of *InputIterator* is *std::size_t*. All indices must refer to vertices already added.

template <class InputIterator>
bool *B.test_facet(InputIterator first, InputIterator beyond)*

returns *true* if a facet described by the vertex indices in the range *[first,beyond)* can be successfully inserted, e.g., with *add_facet(first,beyond)*.
Precondition: The value type of *InputIterator* is *std::size_t*. All indices must refer to vertices already added.

Vertex_handle *B.vertex(std::size_t i)*

returns handle for the vertex of index *i*, or *Vertex_handle* if there is no *i*-th vertex.

bool *B.error() const* returns error status of the builder.

void *B.rollback()* undoes all changes made to the halfedge data structure since the last *begin_surface()* in relative indexing, and deletes the whole surface in absolute indexing. It needs a new call to *begin_surface()* to start inserting again.

bool *B.check_unconnected_vertices()*

returns *true* if unconnected vertices are detected. If *verbose* was set to *true* (see the constructor above) debug information about the unconnected vertices is printed.

bool *B.remove_unconnected_vertices()*

returns *true* if all unconnected vertices could be removed successfully. This happens either if no unconnected vertices had appeared or if the halfedge data structure supports the removal of individual elements.

See Also

CGAL::Polyhedron_3<Traits> . . . . page 1795
HalfedgeDS . . . . page 1843
CGAL::Modifier_base.

Example

A modifier class creates a new triangle in the halfedge data structure using the incremental builder.

```
#include <CGAL/Simple_cartesian.h>
#include <CGAL/Polyhedron_incremental_builder_3.h>
#include <CGAL/Polyhedron_3.h>

// A modifier creating a triangle with the incremental builder.
template <class HDS>
class Build_triangle : public CGAL::Modifier_base<HDS> {
public:
    Build_triangle() {}
    void operator()( HDS& hds) {
        // Postcondition: 'hds' is a valid polyhedral surface.
        CGAL::Polyhedron_incremental_builder_3<HDS> B( hds, true);
        B.begin_surface( 3, 1, 6);
        typedef typename HDS::Vertex   Vertex;
        typedef typename Vertex::Point Point;
        B.add_vertex( Point( 0, 0, 0));
        B.add_vertex( Point( 1, 0, 0));
        B.add_vertex( Point( 0, 1, 0));
        B.begin_facet();
        B.add_vertex_to_facet( 0);
        B.add_vertex_to_facet( 1);
        B.add_vertex_to_facet( 2);
        B.end_facet();
        B.end_surface();
    }
};

typedef CGAL::Simple_cartesian<double>     Kernel;
typedef CGAL::Polyhedron_3<Kernel>         Polyhedron;
typedef Polyhedron::HalfedgeDS             HalfedgeDS;

int main() {
    Polyhedron P;
    Build_triangle<HalfedgeDS> triangle;
    P.delegate( triangle);
    CGAL_assertion( P.is_triangle( P.halfedges_begin()));
    return 0;
}
```

File: examples/Polyhedron/polyhedron_prog_incr_builder.cpp

Concept

PolyhedronItems_3

Definition

The PolyhedronItems_3 concept extends the *HalfedgeDSItems* concept on page 1873. In addition to the requirements stated there, a model for this concept must fulfill the following requirements for the local *PolyhedronItems_3::Vertex_wrapper<Refs,Traits>::Vertex* type and *PolyhedronItems_3::Face_wrapper< Refs,Traits>::Face* type in order to support the point for vertices and the optional plane equation for facets. Note that the items class uses face instead of facet. Only the polyhedral surface renames faces to facets.

Refines

HalfedgeDSItems . . . . page 1854

Types in *PolyhedronItems_3::Vertex_wrapper<Refs,Traits>::Vertex*

Vertex:: Point	point type stored in vertices. A *HalfedgeDS* has no dimension, so this type is named *Point* and not *Point_3*.
Vertex:: Supports_vertex_point	≡ *CGAL::Tag_true*. A point is always required.

Operations

Point&	*v.point()*	
const Point&	*v.point() const*	point.

Types in *PolyhedronItems_3::Face_wrapper<Refs,Traits>::Face*

Types for (optionally) associated geometry in faces. If it is not supported the respective type has to be defined, although it can be an arbitrary dummy type, such as *void** or *Tag_false*.

Face:: Plane	plane type stored in faces. A *HalfedgeDS* has no dimension, so this type is named *Plane* and not *Plane_3*.
Face:: Supports_face_plane	either *CGAL::Tag_true* or *CGAL::Tag_false*.

Operations required if *Supports_face_plane* ≡ *CGAL::Tag_true*

Plane&	*f.plane()*	
const Plane&	*f.plane() const*	plane equation.

Has Models

CGAL::Polyhedron_items_3 . . . . page 1820
CGAL::Polyhedron_min_items_3 . . . . page 1822

See Also

CGAL::Polyhedron_3<Traits> . . . . page 1795
HalfedgeDSItems . . . . page 1854
CGAL::HalfedgeDS_items_2 . . . . page 1873
CGAL::HalfedgeDS_vertex_base<Refs> . . . . page 1883
CGAL::HalfedgeDS_halfedge_base<Refs> . . . . page 1871
CGAL::HalfedgeDS_face_base<Refs> . . . . page 1868

Example

We define our own items class based on the available *CGAL::HalfedgeDS_face_base* base class for faces. We derive the the *Halfedge_wrapper* without further modifications from the *CGAL::HalfedgeDS_items_2*, replace the *Face_wrapper* definition with our new definition, and also replace the *Vertex_wrapper* with a definition that uses *Point_3* instead of *Point_2* as point type. The result is a model for the *PolyhedronItems_3* concept similar to the available *CGAL::Polyhedron_items_3* class. See also there for another illustrative example.

```
#include <CGAL/HalfedgeDS_bases.h>

struct My_items : public CGAL::HalfedgeDS_items_2 {
    template < class Refs, class Traits>
    struct Vertex_wrapper {
        typedef typename Traits::Point_3 Point;
        typedef CGAL::HalfedgeDS_vertex_base< Refs, CGAL::Tag_true,
   Point> Vertex;
    };
    template < class Refs, class Traits>
    struct Face_wrapper {
        typedef typename Traits::Plane_3 Plane;
        typedef CGAL::HalfedgeDS_face_base< Refs, CGAL::Tag_true, Plane>
     Face;
    };
};
```

CGAL::Polyhedron_items_3

Definition

The class *Polyhedron_items_3* is a model of the *PolyhedronItems_3* concept. It provides definitions for vertices with points, halfedges, and faces with plane equations. The polyhedron traits class must provide the respective types for the point and the plane equation. Vertices and facets both contain a halfedge handle to an incident halfedge.

#include <CGAL/Polyhedron_items_3.h>

Is Model for the Concepts

PolyhedronItems_3 . . . . page 1818

Types in *Polyhedron_items_3::Vertex_wrapper<Refs,Traits>::Vertex*

typedef Traits::Point_3	*Point;*
typedef CGAL::Tag_true	*Supports_vertex_point;*

Types in *Polyhedron_items_3::Face_wrapper<Refs,Traits>::Face*

typedef Traits::Plane_3	*Plane;*
typedef CGAL::Tag_true	*Supports_face_plane;*

Creation

Polyhedron_items_3 items;	default constructor.

Operations

Supported as required by the *PolyhedronItems_3* concept.

See Also

CGAL::Polyhedron_3<Traits> . . . . page 1795
CGAL::Polyhedron_min_items_3 . . . . page 1822
CGAL::HalfedgeDS_min_items . . . . page 1879
CGAL::HalfedgeDS_items_2 . . . . page 1873

Example

The following example program defines a new face class based on the *CGAL::HalfedgeDS_face_base* and adds a new color member variable. The new face class is used to replace the face definition in the *CGAL::Polyhedron_items_3* class. The main function illustrates the access to the new member variable. See also the *PolyhedronItems_3* concept for another illustrative example.

```
#include <CGAL/Simple_cartesian.h>
#include <CGAL/IO/Color.h>
#include <CGAL/Polyhedron_3.h>

// A face type with a color member variable.
template <class Refs>
struct My_face : public CGAL::HalfedgeDS_face_base<Refs> {
    CGAL::Color color;
};

// An items type using my face.
struct My_items : public CGAL::Polyhedron_items_3 {
    template <class Refs, class Traits>
    struct Face_wrapper {
        typedef My_face<Refs> Face;
    };
};

typedef CGAL::Simple_cartesian<double>        Kernel;
typedef CGAL::Polyhedron_3<Kernel, My_items>  Polyhedron;
typedef Polyhedron::Halfedge_handle           Halfedge_handle;

int main() {
    Polyhedron P;
    Halfedge_handle h = P.make_tetrahedron();
    h->facet()->color = CGAL::RED;
    return 0;
}
```

File: examples/Polyhedron/polyhedron_prog_color.cpp

Class

CGAL::Polyhedron_min_items_3

Definition

The class *Polyhedron_min_items_3* is a minimal model of the *PolyhedronItems_3* concept. It provides definitions for vertices containing points, halfedges, and faces. The polyhedron traits class must provide the respective type for the point. Vertices and facets both do *not* contain a halfedge handle to an incident halfedge.

#include <CGAL/Polyhedron_min_items_3.h>

Is Model for the Concepts

PolyhedronItems_3 . . . . page 1818

Types in *Polyhedron_min_items_3::Vertex_wrapper<Refs,Traits>::Vertex*

typedef Traits::Point_3	*Point;*
typedef CGAL::Tag_true	*Supports_vertex_point;*

Types in *Polyhedron_min_items_3::Face_wrapper<Refs,Traits>::Face*

typedef CGAL::Tag_false	*Supports_face_plane;*

Creation

Polyhedron_min_items_3 items;	default constructor.

Operations

Supported as required by the *PolyhedronItems_3* concept.

See Also

CGAL::Polyhedron_3<Traits> . . . . page 1795
CGAL::Polyhedron_items_3 . . . . page 1820
CGAL::HalfedgeDS_min_items . . . . page 1879
CGAL::HalfedgeDS_items_2 . . . . page 1873

PolyhedronTraits_3

Required types and member functions for the PolyhedronTraits_3 concept. This geometric traits concept is used in the polyhedral surface data structure *CGAL::Polyhedron_3<Traits>*. This concept is a subset of the 3d kernel traits and any CGAL kernel model can be used directly as template argument.

Refines

CopyConstructable, *Assignable*.

Types

PolyhedronTraits_3:: Point_3 — point type.

PolyhedronTraits_3:: Plane_3 — plane equation. Even if plane equations are not supported with a particular polyhedral surface this type has to be defined (some dummy type).

PolyhedronTraits_3:: Construct_opposite_plane_3

is an unary function object that reverses the plane orientation. Must provide *Plane_3 operator()(Plane_3 plane)* that returns the reversed plane. Required only if plane equations are supported and the *inside_out()* method is used to reverse the polyhedral surface orientation.

Creation

PolyhedronTraits_3 traits(traits2); — copy constructor.

PolyhedronTraits_3& *traits = traits2* — assignment.

Operations

Construct_opposite_plane_3 *traits.construct_opposite_plane_3_object()*

returns an instance of this function object.

Has Models

CGAL::Polyhedron_traits_3<Kernel> . . . . page 1824
CGAL::Polyhedron_traits_with_normals_3<Kernel> . . . . page 1826
All models of the *CGAL::Kernel* concept, e.g., *Simple_cartesian<FieldNumberType>*.

See Also

CGAL::Polyhedron_3<Traits> . . . . page 1795

Class

CGAL::Polyhedron_traits_3<Kernel>

Definition

The class *Polyhedron_traits_3<Kernel>* is a model of the *PolyhedronTraits_3* concept. It defines the geometric types and primitive operations used in the polyhedral surface data structure *CGAL::Polyhedron_3< PolyhedronTraits_3>* in terms of the CGAL *Kernel*. It keeps a local copy of the kernel which makes it suitable for kernels with local state.

#include <CGAL/Polyhedron_traits_3.h>

Is Model for the Concepts

PolyhedronTraits_3 . . . . page 1823

Types

Polyhedron_traits_3<Kernel>:: Kernel		the *Kernel* model.
typedef Kernel::Point_3	*Point_3;*	
typedef Kernel::Plane_3	*Plane_3;*	
typedef Kernel::Construct_opposite_plane_3	*Construct_opposite_plane_3;*	

Creation

Polyhedron_traits_3<Kernel> traits;	default constructor, uses *Kernel()* as local reference to the kernel.
Polyhedron_traits_3<Kernel> traits(Kernel kernel);	stores *kernel* as local reference.

Operations

Construct_opposite_plane_3	*traits.construct_opposite_plane_3_object()*	forwarded to *kernel*.

See Also

CGAL::Polyhedron_traits_with_normals_3<Kernel> . . . . .page 1826

Implementation

Since the *PolyhedronTraits_3* concept is a subset of the 3D kernel concept, this class just forwards the relevant types and access member functions from its template argument. However, it is useful for testing sufficiency of requirements.

Example

Instantiation of a polyhedral surface with the Cartesian kernel based on double coordinates.

```
#include <CGAL/Simple_cartesian.h>
#include <CGAL/Polyhedron_3.h>

typedef CGAL::Simple_cartesian<double>     Kernel;
typedef CGAL::Polyhedron_3<Kernel>         Polyhedron;
typedef Polyhedron::Halfedge_handle        Halfedge_handle;

int main() {
    Polyhedron P;
    Halfedge_handle h = P.make_tetrahedron();
    if ( P.is_tetrahedron(h))
        return 0;
    return 1;
}
```

File: examples/Polyhedron/polyhedron_prog_simple.cpp

CGAL::Polyhedron_traits_with_normals_3<Kernel>

Definition

The class *Polyhedron_traits_with_normals_3<Kernel>* is a model of the *PolyhedronTraits_3* concept. It defines the geometric types and primitive operations used in the polyhedral surface data structure *CGAL::Polyhedron_3<PolyhedronTraits_3>*. *Polyhedron_traits_with_normals_3<Kernel>* uses the normal vector from *Kernel* for the plane equation in facets. It keeps a local copy of the kernel which makes it suitable for kernels with local state.

#include <CGAL/Polyhedron_traits_with_normals_3.h>

Is Model for the Concepts

PolyhedronTraits_3 . . . . page 1823

Types

Polyhedron_traits_with_normals_3<Kernel>:: Kernel		the *Kernel* model.
typedef Kernel::Point_3	*Point_3;*	
typedef Kernel::Vector_3	*Plane_3;*	
typedef Kernel::Construct_opposite_vector_3	*Construct_opposite_plane_3;*	

Creation

Polyhedron_traits_with_normals_3<Kernel> traits;	default constructor, uses *Kernel()* as local reference to the kernel.
Polyhedron_traits_with_normals_3<Kernel> traits(Kernel kernel);	stores *kernel* as local reference.

Operations

Construct_opposite_plane_3 *traits.construct_opposite_plane_3_object()*

forwarded to *kernel*.

See Also

CGAL::Polyhedron_traits_3<Kernel> . . . . page 1824

Example

We use this traits class to instantiate a polyhedral surface with a normal vector and no plane equation for each facet. We compute the normal vector assuming exact arithmetic (integers in this example) and convex planar facets.

```
#include <CGAL/Homogeneous.h>
#include <CGAL/Polyhedron_traits_with_normals_3.h>
#include <CGAL/Polyhedron_3.h>
#include <iostream>
#include <algorithm>

struct Normal_vector {
    template <class Facet>
    typename Facet::Plane_3 operator()( Facet& f) {
        typename Facet::Halfedge_handle h = f.halfedge();
        // Facet::Plane_3 is the normal vector type. We assume the
        // CGAL Kernel here and use its global functions.
        return CGAL::cross_product(
          h->next()->vertex()->point() - h->vertex()->point(),
          h->next()->next()->vertex()->point() -
   h->next()->vertex()->point());
    }
};

typedef CGAL::Homogeneous<int>                                Kernel;
typedef Kernel::Point_3                                       Point_3;
typedef Kernel::Vector_3                                      Vector_3;
typedef CGAL::Polyhedron_traits_with_normals_3<Kernel> Traits;
typedef CGAL::Polyhedron_3<Traits>                            Polyhedron;

int main() {
    Point_3 p( 1, 0, 0);
    Point_3 q( 0, 1, 0);
    Point_3 r( 0, 0, 1);
    Point_3 s( 0, 0, 0);
    Polyhedron P;
    P.make_tetrahedron( p, q, r, s);
    std::transform( P.facets_begin(), P.facets_end(), P.planes_begin(),
                    Normal_vector());
    CGAL::set_pretty_mode( std::cout);
    std::copy( P.planes_begin(), P.planes_end(),
               std::ostream_iterator<Vector_3>( std::cout, "\n"));
    return 0;
}
```

File: examples/Polyhedron/polyhedron_prog_normals.cpp

Function

CGAL::operator<<

Definition

This operator writes the polyhedral surface *P* to the output stream *out* using the Object File Format, OFF, with file extension `.off`, which is also understood by GeomView [Phi96]. The output is in ASCII format. From the polyhedral surface, only the point coordinates and facets are written. Neither normal vectors nor color attributes are used.

For OFF an ASCII and a binary format exist. The format can be selected with the CGAL modifiers for streams, *set_ascii_mode* and *set_binary_mode* respectively. The modifier *set_pretty_mode* can be used to allow for (a few) structuring comments in the output. Otherwise, the output would be free of comments. The default for writing is ASCII without comments.

#include <CGAL/IO/Polyhedron_iostream.h>

template <class PolyhedronTraits_3>
ostream& *ostream& out << CGAL::Polyhedron_3<PolyhedronTraits_3> P*

See Also

CGAL::Polyhedron_3<Traits> . . . . page 1795
operator>> . . . . page 1829

CGAL::operator>>

Definition

This operator reads a polyhedral surface in Object File Format, OFF, with file extension `.off`, which is also understood by Geomview [Phi96], from the input stream *in* and appends it to the polyhedral surface *P*. Only the point coordinates and facets from the input stream are used to build the polyhedral surface. Neither normal vectors nor color attributes are evaluated. If the stream *in* does not contain a permissible polyhedral surface the *ios::badbit* of the input stream *in* is set and *P* remains unchanged.

For OFF an ASCII and a binary format exist. The stream detects the format automatically and can read both.

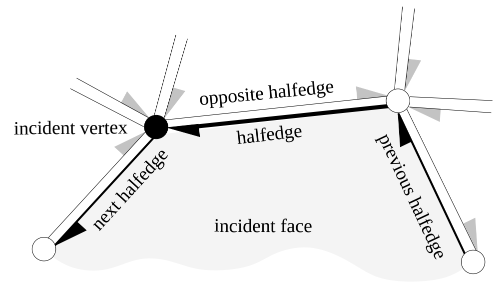
#include <CGAL/IO/Polyhedron_iostream.h>

template <class PolyhedronTraits_3>
istream& *istream& in >> CGAL::Polyhedron_3<PolyhedronTraits_3>& P*

See Also

Implementation

This operator is implemented using the modifier mechanism for polyhedral surfaces and the *CGAL::Polyhedron_incremental_builder_3* class, which allows the construction in a single, efficient scan pass of the input and handles also all the possible flexibility of the polyhedral surface.



Chapter 26

Halfedge Data Structures

Lutz Kettner

Contents

26.1 Introduction

A halfedge data structure (abbreviated as *HalfedgeDS*, or *HDS* for template parameters) is an edge-centered data structure capable of maintaining incidence information of vertices, edges and faces, for example for planar maps, polyhedra, or other orientable, two-dimensional surfaces embedded in arbitrary dimension. Each edge is decomposed into two halfedges with opposite orientations. One incident face and one incident vertex are stored in each halfedge. For each face and each vertex, one incident halfedge is stored. Reduced variants of the halfedge data structure can omit some of these information, for example the halfedge pointers in faces or the storage of faces at all.

The halfedge data structure is a combinatorial data structure, geometric interpretation is added by classes built on top of the halfedge data structure. These classes might be more convenient to use than the halfedge data structure directly, since the halfedge data structure is meant as an implementation layer. See for example the *CGAL::Polyhedron_3* class in Chapter 25.

The data structure provided here is also known as the FE-structure [Wei85], as halfedges [Män88, BFH95] or as the doubly connected edge list (DCEL) [dBvKOS97], although the original reference for the DCEL [MP78]

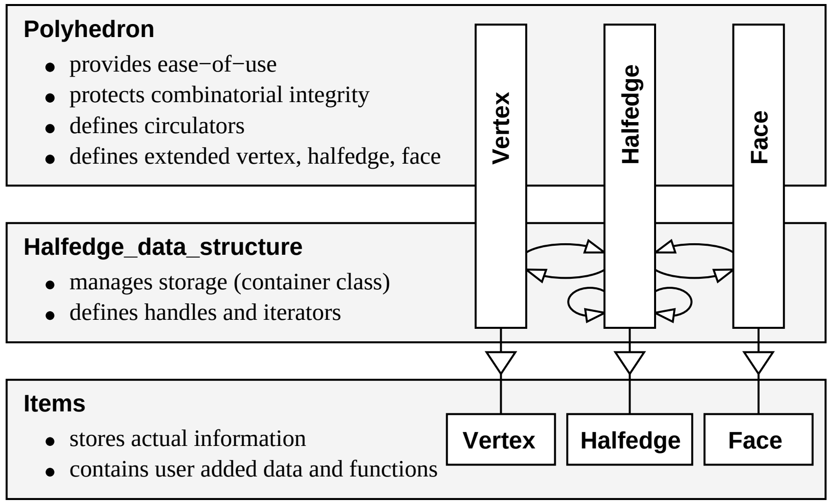


Figure 26.1: Responsibilities of the different layers in the halfedge data-structure design.

describes a different data structure. The halfedge data structure can also be seen as one of the variants of the quad-edge data structure [GS85]. In general, the quad-edge data can represent non-orientable 2-manifolds, but the variant here is restricted to orientable 2-manifolds only. An overview and comparison of these different data structures together with a thorough description of the design implemented here can be found in [Ket99].

The design presented here is a revised and incompatible version of the previous design [Ket98] as used in CGAL R2.2 and earlier releases. Files and identifier names are disjoint with the old design which allows for both versions to co-exists. However, classes using a halfedge data structure can only use one design. For example the polyhedral surface *Polyhedron_3* uses by default the new design. See Chapter 25 for how to still select the old implementation.

26.2 Software Design

Figure 26.1 illustrates the responsibilities of the three layers of the software design, with the *CGAL::Polyhedron_3* as an example for the top layer. The items provide the space for the information that is actually stored, i.e., with member variables and access member functions in *Vertex*, *Halfedge*, and *Face* respectively. Halfedges are required to provide a reference to the next halfedge and to the opposite halfedge. Optionally they may provide a reference to the previous halfedge, to the incident vertex, and to the incident face. Vertices and faces may be empty. Optionally they may provide a reference to the incident halfedge. The options mentioned are supported in the halfedge data structure and the polyhedron, for example, Euler operations update the optional references if they are present. Furthermore, the item classes can be extended with arbitrary attributes and member functions, which will be promoted by inheritance to the actual classes used for the polyhedron.

Vertices, halfedges, and faces are passed as local types of the *Items* class to the halfedge data structure and polyhedron. Implementations for vertices, halfedges and faces are provided that fulfill the mandatory part of the requirements. They can be used as base classes for extensions by the user. Richer implementations are also provided to serve as defaults; for polyhedra they provide all optional incidences, a three-dimensional point in the vertex type and a plane equation in the face type.

The *Halfedge_data_structure*, concept *HalfedgeDS*, is responsible for the storage organization of the items. Currently, implementations using internally a bidirectional list or a vector are provided. The *HalfedgeDS* defines the handles and iterators belonging to the items. These types are promoted to the declaration of the items themselves and are used there to provide the references to the incident items. This promotion of types is done with a template parameter *Refs* of the item types. The halfedge data structure provides member functions to insert and delete items, to traverse all items, and it gives access to the items.

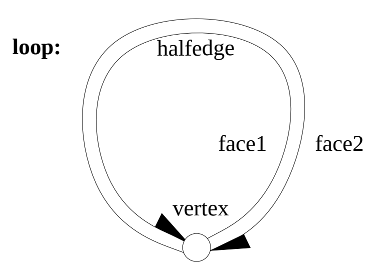
There are two different models for the *HalfedgeDS* concept available, *HalfedgeDS_list* and *HalfedgeDS_vector*, and more might come. Therefore we have kept their interface small and factored out common functionality into separate helper classes, *HalfedgeDS_decorator*, *HalfedgeDS_const_decorator*, and *HalfedgeDS_items_decorator*, which are not shown in Figure 26.1, but would be placed at the side of the *HalfedgeDS* since they broaden that interface but do not hide it. These helper classes contain operations that are useful to implement the operations in the next layer, for example, the polyhedron. They add, for example, the Euler operations and partial operations from which further Euler operations can be built, such as inserting an edge into the ring of edges at a vertex. Furthermore, the helper classes contain adaptive functionality. For example, if the *prev()* member function is not provided for halfedges, the *find_prev()* member function of a helper class searches in the positive direction along the face for the previous halfedge. But if the *prev()* member function is provided, the *find_prev()* member function simply calls it. This distinction is resolved at compile time with a technique called *compile-time tags*, similar to iterator tags in [SL95].

The *Polyhedron_3* as an example for the third layer adds the geometric interpretation, provides an easy-to-use interface of high-level functions, and unifies the access to the flexibility provided underneath. It renames face to facet, which is more common for three-dimensional surfaces. The interface is designed to protect the integrity of the internal representation, the handles stored in the items can no longer directly be written by the user. The polyhedron adds the convenient and efficient circulators, see *Circulator*, for accessing the circular sequence of edges around a vertex or around a facet. To achieve this, the *Polyhedron_3* derives new vertices, halfedges and facets from those provided in *Items*. These new items are those actually used in the *HalfedgeDS*, which gives us the coherent type structure in this design, especially if compared to our previous design.

26.3 Example Programs

26.3.1 The Default Halfedge Data Structure

The following example program uses the default halfedge data structure and the decorator class. The default halfedge data structure uses a list-based representation. All incidences of the items and a point type for vertices are defined. The trivial traits class provides the type used for the point. The program creates a loop, consisting of two halfedges, one vertex and two faces, and checks its validity.




```
#include <CGAL/HalfedgeDS_default.h>
#include <CGAL/HalfedgeDS_decorator.h>

struct Traits { typedef int Point_2; };
typedef CGAL::HalfedgeDS_default<Traits> HDS;
typedef CGAL::HalfedgeDS_decorator<HDS> Decorator;
```

```
int main() {
    HDS hds;
    Decorator decorator(hds);
    decorator.create_loop();
    CGAL_assertion( decorator.is_valid());
    return 0;
}
```

File: examples/HalfedgeDS/hds_prog_default.cpp

26.3.2 A Minimal Halfedge Data Structure

The following program defines a minimal halfedge data structure using the minimal items class *CGAL::HalfedgeDS_min_items* and a list-based halfedge data structure. The result is a data structure maintaining only halfedges with next and opposite pointers. No vertices or faces are stored. The data structure represents an *undirected graph*.

```
#include <CGAL/HalfedgeDS_min_items.h>
#include <CGAL/HalfedgeDS_default.h>
#include <CGAL/HalfedgeDS_decorator.h>

// no traits needed, argument can be arbitrary dummy.
typedef CGAL::HalfedgeDS_default<int, CGAL::HalfedgeDS_min_items> HDS;
typedef CGAL::HalfedgeDS_decorator<HDS>  Decorator;

int main() {
    HDS hds;
    Decorator decorator(hds);
    decorator.create_loop();
    CGAL_assertion( decorator.is_valid());
    return 0;
}
```

File: examples/HalfedgeDS/hds_prog_graph.cpp

26.3.3 The Default with a Vector Instead of a List

The default halfedge data structure uses a list internally and the maximal base classes. We change the list to a vector representation here. Again, a trivial traits class provides the type used for the point. Note that for the vector storage the size of the halfedge data structure should be reserved beforehand, either with the constructor as shown in the example or with the *reserve()* member function. One can later resize the data structure with further calls to the *reserve()* member function, but only if the data structure is in a consistent, i.e., *valid*, state.

```
#include <CGAL/HalfedgeDS_items_2.h>
#include <CGAL/HalfedgeDS_vector.h>
#include <CGAL/HalfedgeDS_decorator.h>

struct Traits { typedef int Point_2; };
typedef CGAL::HalfedgeDS_vector< Traits, CGAL::HalfedgeDS_items_2> HDS;
```

```
typedef CGAL::HalfedgeDS_decorator<HDS>  Decorator;

int main() {
    HDS hds(1,2,2);
    Decorator decorator(hds);
    decorator.create_loop();
    CGAL_assertion( decorator.is_valid());
    return 0;
}
```

File: examples/HalfedgeDS/hds_prog_vector.cpp

26.3.4 Example Adding Color to Faces

This example re-uses the base class available for faces and adds a member variable *color*.

```
#include <CGAL/HalfedgeDS_items_2.h>
#include <CGAL/HalfedgeDS_default.h>
#include <CGAL/IO/Color.h>

// A face type with a color member variable.
template <class Refs>
struct My_face : public CGAL::HalfedgeDS_face_base<Refs> {
    CGAL::Color color;
    My_face() {}
    My_face( CGAL::Color c) : color(c) {}
};

// An items type using my face.
struct My_items : public CGAL::HalfedgeDS_items_2 {
    template <class Refs, class Traits>
    struct Face_wrapper {
        typedef My_face<Refs> Face;
    };
};

struct My_traits { // arbitrary point type, not used here.
    typedef int  Point_2;
};

typedef CGAL::HalfedgeDS_default<My_traits, My_items> HDS;
typedef HDS::Face                                     Face;
typedef HDS::Face_handle                              Face_handle;

int main() {
    HDS hds;
    Face_handle f = hds.faces_push_back( Face( CGAL::RED));
    f->color = CGAL::BLUE;
    CGAL_assertion( f->color == CGAL::BLUE);
    return 0;
}
```

File: examples/HalfedgeDS/hds_prog_color.cpp

26.3.5 Example Defining a More Compact Halfedge

advanced

The halfedge data structure as presented here is slightly less space efficient as, for example, the winged-edge data structure [Bau75], the DCEL [MP78] or variants of the quad-edge data structure [GS85]. On the other hand, it does not require any search operations during traversals. A comparison can be found in [Ket99].

The following example trades traversal time for a compact storage representation using traditional C techniques (i.e., type casting and the assumption that pointers, especially those from malloc or new, point to even addresses). The idea goes as follows: The halfedge data structure allocates halfedges pairwise. Concerning the vector-based data structure this implies that the absolute value of the difference between a halfedge and its opposite halfedge is always one with respect to C pointer arithmetic. We can replace the opposite pointer by a single bit encoding the sign of this difference. We will store this bit as the least significant bit in the next halfedge handle. Furthermore, we do not implement a pointer to the previous halfedge. What remains are three pointers per halfedge.

We use the static member function *halfedge_handle()* to convert from pointers to halfedge handles. The same solution can be applied to the list-based halfedge data structure *CGAL::HalfedgeDS_list*, see examples/HalfedgeDS/hds_prog_compact2.cpp. Here is the example for the vector-based data structure.

```
#include <CGAL/HalfedgeDS_items_2.h>
#include <CGAL/HalfedgeDS_vector.h>
#include <CGAL/HalfedgeDS_decorator.h>
#include <cstddef>

// Define a new halfedge class. We assume that the Halfedge_handle can
// be created from a pointer (e.g. the HalfedgeDS is based here on the
// In_place_list or a std::vector with such property) and that halfedges
// are allocated in pairs. We encode the opposite pointer in a single
   bit,
// which is stored in the lower bit of the next-pointer. We use the
// static member function HDS::halfedge_handle to translate pointer to
// handles.
template <class Refs>
class My_halfedge {
public:
    typedef Refs                                    HDS;
    typedef My_halfedge<Refs>                       Base_base;
    typedef My_halfedge<Refs>                       Base;
    typedef My_halfedge<Refs>                       Self;
    typedef CGAL::Tag_false                         Supports_halfedge_prev;
    typedef CGAL::Tag_true
   Supports_halfedge_vertex;
    typedef CGAL::Tag_true                          Supports_halfedge_face;
    typedef typename Refs::Vertex_handle            Vertex_handle;
    typedef typename Refs::Vertex_const_handle      Vertex_const_handle;
    typedef typename Refs::Halfedge                 Halfedge;
    typedef typename Refs::Halfedge_handle          Halfedge_handle;
    typedef typename Refs::Halfedge_const_handle Halfedge_const_handle;
    typedef typename Refs::Face_handle              Face_handle;
```

```
    typedef typename Refs::Face_const_handle     Face_const_handle;
private:
    std::ptrdiff_t  nxt;
public:
    My_halfedge() : nxt(0), f( Face_handle()) {}

    Halfedge_handle opposite() {
        // Halfedge could be different from My_halfedge (e.g. pointer for
        // linked list). Get proper handle from 'this' pointer first, do
        // pointer arithmetic, then convert pointer back to handle again.
        Halfedge_handle h = HDS::halfedge_handle(this); // proper handle
        if ( nxt & 1)
            return HDS::halfedge_handle( &* h + 1);
        return HDS::halfedge_handle( &* h - 1);
    }
    Halfedge_const_handle opposite() const { // same as above
        Halfedge_const_handle h = HDS::halfedge_handle(this); // proper
   handle
        if ( nxt & 1)
            return HDS::halfedge_handle( &* h + 1);
        return HDS::halfedge_handle( &* h - 1);
    }
    Halfedge_handle next() {
        return HDS::halfedge_handle((Halfedge*)(nxt & (~
   std::ptrdiff_t(1))));
    }
    Halfedge_const_handle next() const {
        return HDS::halfedge_handle((const Halfedge*)
                                    (nxt & (~ std::ptrdiff_t(1))));
    }
    void  set_opposite( Halfedge_handle h) {
        CGAL_precondition(( &* h - 1 == &* HDS::halfedge_handle(this)) ||
                          ( &* h + 1 == &* HDS::halfedge_handle(this)));
        if ( &* h - 1 == &* HDS::halfedge_handle(this))
            nxt |= 1;
        else
            nxt &= (~ std::ptrdiff_t(1));
    }
    void  set_next( Halfedge_handle h) {
        CGAL_precondition( ((std::ptrdiff_t)(&*h) & 1) == 0);
        nxt = ((std::ptrdiff_t)(&*h)) | (nxt & 1);
    }
private:    // Support for the Vertex_handle.
    Vertex_handle     v;
public:
    // the incident vertex.
    Vertex_handle          vertex()                      { return v; }
    Vertex_const_handle    vertex() const                { return v; }
    void                   set_vertex( Vertex_handle w) { v = w; }

private:
    Face_handle      f;
public:
    Face_handle            face()                        { return f; }
```

```
    Face_const_handle      face() const                 { return f; }
    void                   set_face( Face_handle g)     { f = g; }
    bool                   is_border() const { return f == Face_handle();
   }
};

// Replace halfedge in the default items type.
struct My_items : public CGAL::HalfedgeDS_items_2 {
    template <class Refs, class Traits>
    struct Halfedge_wrapper {
        typedef My_halfedge<Refs> Halfedge;
    };
};

struct Traits { typedef int Point_2; };
typedef CGAL::HalfedgeDS_vector<Traits, My_items> HDS;
typedef CGAL::HalfedgeDS_decorator<HDS>  Decorator;

int main() {
    HDS hds(1,2,2);
    Decorator decorator(hds);
    decorator.create_loop();
    CGAL_assertion( decorator.is_valid());
    return 0;
}
```

File: examples/HalfedgeDS/hds_prog_compact.cpp

advanced

26.3.6 Example Using the Halfedge Iterator

Two edges are created in the default halfedge data structure. The halfedge iterator is used to count the halfedges.

```
#include <CGAL/HalfedgeDS_default.h>
#include <CGAL/HalfedgeDS_decorator.h>

struct Traits { typedef int Point_2; };
typedef CGAL::HalfedgeDS_default<Traits> HDS;
typedef CGAL::HalfedgeDS_decorator<HDS>  Decorator;
typedef HDS::Halfedge_iterator           Iterator;

int main() {
    HDS hds;
    Decorator decorator(hds);
    decorator.create_loop();
    decorator.create_segment();
    CGAL_assertion( decorator.is_valid());
    int n = 0;
    for ( Iterator i = hds.halfedges_begin(); i != hds.halfedges_end();
   ++i )
        ++n;
```

```
    CGAL_assertion( n == 4);  // == 2 edges
    return 0;
}
```

File: examples/HalfedgeDS/hds_prog_halfedge_iterator.cpp

26.3.7 Example for an Adapter to Build an Edge Iterator

Three edges are created in the default halfedge data structure. The adapter *N_step_adaptor* is used to declare the edge iterator used in counting the edges.

```
#include <CGAL/HalfedgeDS_default.h>
#include <CGAL/HalfedgeDS_decorator.h>
#include <CGAL/N_step_adaptor.h>

struct Traits { typedef int Point_2; };
typedef CGAL::HalfedgeDS_default<Traits>            HDS;
typedef CGAL::HalfedgeDS_decorator<HDS>             Decorator;
typedef HDS::Halfedge_iterator                      Halfedge_iterator;
typedef CGAL::N_step_adaptor< Halfedge_iterator, 2> Iterator;

int main() {
    HDS hds;
    Decorator decorator(hds);
    decorator.create_loop();
    decorator.create_segment();
    CGAL_assertion( decorator.is_valid());
    int n = 0;
    for ( Iterator e = hds.halfedges_begin(); e != hds.halfedges_end();
   ++e)
        ++n;
    CGAL_assertion( n == 2);  // == 2 edges
    return 0;
}
```

File: examples/HalfedgeDS/hds_prog_edge_iterator.cpp

Halfedge Data Structure Reference Manual

Lutz Kettner

A halfedge data structure (abbreviated as *HalfedgeDS*, or *HDS* for template parameters) is an edge-centered data structure capable of maintaining incidence information of vertices, edges and faces, for example for planar maps or polyhedral surfaces. It is a combinatorial data structure, geometric interpretation is added by classes built on top of the halfedge data structure. These classes might be more convenient to use than the halfedge data structure directly, since the halfedge data structure is meant as an implementation layer. See for example the *CGAL::Polyhedron_3* class in Chapter 25.6.2.

The data structure provided here is known as the FE-structure [Wei85], as halfedges [Män88, BFH95] or as the doubly connected edge list (DCEL) [dBvKOS97], although the original reference for the DCEL [MP78] describes a related but different data structure. The halfedge data structure can also be seen as one of the variants of the quad-edge data structure [GS85]. In general, the quad-edge data can represent non-orientable 2-manifolds, but the variant here is restricted to orientable 2-manifolds only. An overview and comparison of these different data structures together with a thorough description of the design implemented here can be found in [Ket99].

26.4 Classified Reference Pages

Concepts

Classes

26.5 Alphabetical List of Reference Pages

HalfedgeDS<Traits,Items,Alloc>

Definition

The concept of a halfedge data structure (abbreviated as *HalfedgeDS*, or *HDS* for template parameters) defines an edge-centered data structure capable of maintaining incidence information of vertices, edges, and faces, for example for planar maps or polyhedral surfaces. It is a combinatorial data structure, geometric interpretation is added by classes built on top of the halfedge data structure.

The data structure defined here is known as the FE-structure [Wei85], as halfedges [Män88, BFH95] or as the doubly connected edge list (DCEL) [dBvKOS97], although the original reference for the DCEL [MP78] describes a different data structure. The halfedge data structure can also be seen as one of the variants of the quad-edge data structure [GS85]. In general, the quad-edge data can represent non-orientable 2-manifolds, but the variant here is restricted to orientable 2-manifolds only. An overview and comparison of these different data structures together with a thorough description of the design implemented here can be found in [Ket99].

Each edge is represented by two halfedges with opposite orientations. Each halfedge can store a reference to an incident face and an incident vertex. For each face and each vertex an incident halfedge is stored. Reduced variants of the halfedge data structure can omit some of these incidences, for example the reference to halfedges in vertices or the storage of vertices at all. See Figure 26.2 for the incidences, the mandatory and optional member functions possible for vertices, halfedges, and faces.

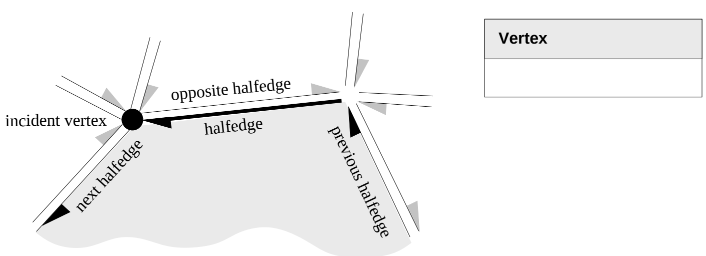


Figure 26.2: The three classes *Vertex*, *Halfedge*, and *Face* of the halfedge data structure. Member functions with shaded background are mandatory. The others are optionally supported.

A HalfedgeDS<Traits,Items,Alloc> organizes the internal storage of its items. Examples are a list-based or a vector-based storage. The HalfedgeDS<Traits,Items,Alloc> exhibits most of the characteristics of the container class used internally, for example the iterator category. A vector resizes automatically when a new item exceeds the reserved space. Since resizing is an expensive operation for a HalfedgeDS<Traits,Items,Alloc> in general and only possible in a well defined state of the data structure (no dangling handles), it must be called explicitly in advance for a HalfedgeDS<Traits,Items,Alloc> before inserting new items beyond the current capacity. Classes built on top of a HalfedgeDS<Traits,Items,Alloc> are advised to call the *reserve()* member function before creating new items.

Parameters

A HalfedgeDS<Traits,Items,Alloc> is a class template and will be used as argument for other class templates, for example *CGAL::Polyhedron_3*. The template parameters to instantiate the HalfedgeDS<Traits,Items,Alloc> will be provided by this other class template. Therefore, the three template parameters and their meaning are mandatory. We distinguish between the template HalfedgeDS<Traits,Items,Alloc> and an instantiation of it.

Traits is a traits class that will be passed to the item types in *Items*. It will not be used in HalfedgeDS< Traits,Items,Alloc> itself. *Items* is a model of the *HalfedgeDSItems* concept. *Alloc* is an allocator that fulfills all requirements of allocators for STL container classes. The *rebind* mechanism from *Alloc* will be used to create appropriate allocators internally. A default argument is mandatory for *Alloc*, for example, the macro *CGAL_ALLOCATOR(int)* from the <*CGAL/memory.h*> header file can be used as default allocator.

Types

HalfedgeDS<Traits,Items,Alloc>:: Traits	traits class.
HalfedgeDS<Traits,Items,Alloc>:: Items	model of *HalfedgeDSItems* concept.
HalfedgeDS<Traits,Items,Alloc>:: size_type	size type.
HalfedgeDS<Traits,Items,Alloc>:: difference_type	difference type.
HalfedgeDS<Traits,Items,Alloc>:: iterator_category	iterator category for all iterators.
HalfedgeDS<Traits,Items,Alloc>:: allocator_type	allocator type *Alloc*.
HalfedgeDS<Traits,Items,Alloc>:: Vertex	model of *HalfedgeDSVertex* concept.
HalfedgeDS<Traits,Items,Alloc>:: Halfedge	model of *HalfedgeDSHalfedge* concept.
HalfedgeDS<Traits,Items,Alloc>:: Face	model of *HalfedgeDSFace* concept.

The following handles and iterators have appropriate non-mutable counterparts, i.e., *const_handle* and *const_iterator*. The mutable types are assignable to their non-mutable counterparts. The iterators are assignable to the respective handle types. Wherever the handles appear in function parameter lists, the corresponding iterators can be used as well. **Note:** The handle types must have a default constructor that creates a unique and always the same handle value. It will be used in analogy to *NULL* for pointers.

HalfedgeDS<Traits,Items,Alloc>:: Vertex_handle	handle to vertex.
HalfedgeDS<Traits,Items,Alloc>:: Halfedge_handle	handle to halfedge.
HalfedgeDS<Traits,Items,Alloc>:: Face_handle	handle to face.
HalfedgeDS<Traits,Items,Alloc>:: Vertex_iterator	iterator over all vertices.
HalfedgeDS<Traits,Items,Alloc>:: Halfedge_iterator	iterator over all halfedges.
HalfedgeDS<Traits,Items,Alloc>:: Face_iterator	iterator over all faces.

advanced

Types for Tagging Optional Features

The following types are equal to either *CGAL::Tag_true* or *CGAL::Tag_false*, depending on whether the named feature is supported or not.

HalfedgeDS<Traits,Items,Alloc>:: Supports_vertex_halfedge	*Vertex::halfedge()*.
HalfedgeDS<Traits,Items,Alloc>:: Supports_halfedge_prev	*Halfedge::prev()*.
HalfedgeDS<Traits,Items,Alloc>:: Supports_halfedge_vertex	*Halfedge::vertex()*.
HalfedgeDS<Traits,Items,Alloc>:: Supports_halfedge_face	*Halfedge::face()*.
HalfedgeDS<Traits,Items,Alloc>:: Supports_face_halfedge	*Face::halfedge()*.
HalfedgeDS<Traits,Items,Alloc>:: Supports_removal	removal of individual elements.

The following dependencies among these options must be regarded:

Vertices are supported $\Longleftrightarrow$ *Supports_halfedge_vertex* $\equiv$ *CGAL::Tag_true*.
Faces are supported $\Longleftrightarrow$ *Supports_halfedge_face* $\equiv$ *CGAL::Tag_true*.
Supports_vertex_halfedge $\equiv$ *CGAL::Tag_true* $\Longrightarrow$ *Supports_halfedge_vertex* $\equiv$ *CGAL::Tag_true*.
Supports_vertex_point $\equiv$ *CGAL::Tag_true* $\Longrightarrow$ *Supports_halfedge_vertex* $\equiv$ *CGAL::Tag_true*.
Supports_face_halfedge $\equiv$ *CGAL::Tag_true* $\Longrightarrow$ *Supports_halfedge_face* $\equiv$ *CGAL::Tag_true*.

advanced

advanced

Static Member Functions

When writing an items type, such as a user defined vertex, certain functions need to create a handle but knowing only a pointer, for example, the *this*-pointer. The following static member functions of HalfedgeDS< Traits,Items,Alloc> create such a corresponding handle for an item type from a pointer. This conversion encapsulates possible adjustments for hidden data members in the true item type, such as linked-list pointers. Note that the user provides item types with the *Items* template argument, which may differ from the *Vertex*, *Halfedge*, and *Face* types defined in HalfedgeDS<Traits,Items,Alloc>. If they differ, they are derived from the user provided item types. We denote the user item types with *Vertex_base*, *Halfedge_base*, and *Face_base* in the following. The fully qualified name for *Vertex_base* would be for example – assuming that the type *Self* refers to the instantiated *HalfedgeDS* –

```
typedef typename Items::template Vertex_wrapper<Self,Traits> Vertex_wrapper;
typedef typename Vertex_wrapper::Vertex Vertex_base;
```

Implementing these functions relies on the fundamental assumption that an iterator (or handle) of the internally used container class can be constructed from a pointer of a contained item only. This is true and controlled by us for *CGAL::In_place_list*. It is true for the *std::vector* of major STL distributions, but not necessarily guaranteed. We might switch to an internal implementation if need arises.

static Vertex_handle	*HalfedgeDS::vertex_handle(Vertex_base* v)*
static Vertex_const_handle	*HalfedgeDS::vertex_handle(const Vertex_base* v)*
static Halfedge_handle	*HalfedgeDS::halfedge_handle(Halfedge_base* h)*
static Halfedge_const_handle	*HalfedgeDS::halfedge_handle(const Halfedge_base* h)*
static Face_handle	*HalfedgeDS::face_handle(Face_base* f)*
static Face_const_handle	*HalfedgeDS::face_handle(const Face_items* f)*

advanced

Creation

HalfedgeDS<Traits,Items,Alloc> hds; — empty halfedge data structure.

HalfedgeDS<Traits,Items,Alloc> hds(size_type v, size_type h, size_type f);

storage reserved for v vertices, h halfedges, and f faces.

HalfedgeDS<Traits,Items,Alloc> hds(hds2); — copy constructor.
Precondition: *hds2* contains no dangling handles.

HalfedgeDS<Traits,Items,Alloc>& $hds = hds2$ — assignment operator.
Precondition: *hds2* contains no dangling handles.

void *hds.reserve(size_type v, size_type h, size_type f)*

reserves storage for v vertices, h halfedges, and f faces. If all capacities are already greater or equal than the requested sizes nothing happens. Otherwise, *hds* will be resized and all handles, iterators and circulators invalidate.
Precondition: If resizing is necessary *hds* contains no dangling handles.

Access Member Functions

Size	*hds.size_of_vertices() const*	number of vertices.
Size	*hds.size_of_halfedges() const*	number of halfedges.
Size	*hds.size_of_faces() const*	number of faces.
Size	*hds.capacity_of_vertices() const*	space reserved for vertices.
Size	*hds.capacity_of_halfedges() const*	space reserved for halfedges.
Size	*hds.capacity_of_faces() const*	space reserved for faces.
size_t	*hds.bytes() const*	bytes used for *hds*.
size_t	*hds.bytes_reserved() const*	bytes reserved for *hds*.
allocator_type	*hds.get_allocator() const*	allocator object.

The following member functions return the non-mutable iterator if *hds* is declared const.

Vertex_iterator	*hds.vertices_begin()*	iterator over all vertices.
Vertex_iterator	*hds.vertices_end()*	
Halfedge_iterator	*hds.halfedges_begin()*	iterator over all halfedges

Halfedge_iterator	*hds.halfedges_end()*	
Face_iterator	*hds.faces_begin()*	iterator over all faces.
Face_iterator	*hds.faces_end()*	

Insertion

Note that the vertex-related and the face-related member functions may not be provided for a HalfedgeDS< Traits,Items,Alloc> that does not support vertices or faces respectively.

Vertex_handle *hds.vertices_push_back(const Vertex& v)*

appends a copy of v to *hds*. Returns a handle of the new vertex.

Halfedge_handle *hds.edges_push_back(const Halfedge& h, const Halfedge& g)*

appends a copy of h and a copy of g to *hds* and makes them opposite to each other. Returns a handle of the copy of h.

Halfedge_handle *hds.edges_push_back(const Halfedge& h)*

appends a copy of h and a copy of $h->opposite()$ to *hds* and makes them opposite to each other. Returns a handle of the copy of h.
Precondition: *h->opposite()* denotes a halfedge.

Face_handle *hds.faces_push_back(const Face& f)*

appends a copy of f to *hds*. Returns a handle of the new face.

Removal

Erasing single elements is optional and indicated with the type tag *Supports_removal*. The *pop_back* and the *clear* member functions are mandatory. If vertices or faces are not supported for a HalfedgeDS< Traits,Items,Alloc> the *pop_back* and the *clear* member functions must be provided as null operations.

void	*hds.vertices_pop_front()*	removes the first vertex if vertices are supported and *Supports_removal* $\equiv$ *CGAL::Tag_true*.
void	*hds.vertices_pop_back()*	removes the last vertex.
void	*hds.vertices_erase(Vertex_handle v)*	removes the vertex v if vertices are supported and *Supports_removal* $\equiv$ *CGAL::Tag_true*.

void *hds.vertices_erase(Vertex_handle first, Vertex_handle last)*

removes the range of vertices $[first, last)$ if vertices are supported and *Supports_removal* $\equiv$ *CGAL::Tag_true*.

void	*hds.edges_pop_front()*	removes the first two halfedges if *Supports_removal* $\equiv$ *CGAL::Tag_true*.
void	*hds.edges_pop_back()*	removes the last two halfedges.
void	*hds.edges_erase(Halfedge_handle h)*	removes the pair of halfedges *h* and *h->opposite()* if *Supports_removal* $\equiv$ *CGAL::Tag_true*.
void	*hds.edges_erase(Halfedge_handle first, Halfedge_handle last)*	
		removes the range of edges $[first, last)$ if *Supports_removal* $\equiv$ *CGAL::Tag_true*.
void	*hds.faces_pop_front()*	removes the first face if faces are supported and *Supports_removal* $\equiv$ *CGAL::Tag_true*.
void	*hds.faces_pop_back()*	removes the last face.
void	*hds.faces_erase(Face_handle f)*	removes the face *f* if faces are supported and *Supports_removal* $\equiv$ *CGAL::Tag_true*.
void	*hds.faces_erase(Face_handle first, Face_handle last)*	
		removes the range of faces $[first, last)$ if faces are supported and *Supports_removal* $\equiv$ *CGAL::Tag_true*.
void	*hds.vertices_clear()*	removes all vertices.
void	*hds.edges_clear()*	removes all halfedges.
void	*hds.faces_clear()*	removes all faces.
void	*hds.clear()*	removes all elements.

advanced

Operations with Border Halfedges

The following notion of *border halfedges* is particular useful where the halfedge data structure is used to model surfaces with boundary, i.e., surfaces with missing faces or open regions. Halfedges incident to an open region are called *border halfedges*. A halfedge is a *border edge* if the halfedge itself or its opposite halfedge is a border halfedge. The only requirement to work with border halfedges is that the *Halfedge* class provides a member function *is_border()* returning a *bool*. Usually, the halfedge data structure supports faces and the value of the default constructor of the face handle will indicate a border halfedge, but this may not be the only possibility. The *is_border()* predicate divides the edges into two classes, the border edges and the non-border edges. The following normalization reorganizes the sequential storage of the edges such that the non-border edges precede the border edges, and that for each border edge the latter of the two halfedges is a border halfedge (the first one might be a border halfedge too). The normalization stores the number of border halfedges, as well as the halfedge iterator where the border edges start at, within the halfedge data structure. These values will be invalid after further halfedge insertions or removals and changes in the border status of a halfedge. There is no automatic update required.

void *hds.normalize_border()*

sorts halfedges such that the non-border edges precede the border edges. For each border edge that is incident to a face, the halfedge iterator will reference the halfedge incident to the face right before the halfedge incident to the open region.

Size *hds.size_of_border_halfedges() const*

number of border halfedges. An edge with no incident face counts as two border halfedges.
Precondition: *normalize_border()* has been called and no halfedge insertion or removal and no change in border status of the halfedges have occurred since then.

Size *hds.size_of_border_edges() const*

number of border edges. If *size_of_border_edges()* is equal to *size_of_border_halfedges()* all border edges are incident to a face on one side and to an open region on the other side.
Precondition: *normalize_border()* has been called and no halfedge insertion or removal and no change in border status of the halfedges have occurred since then.

Halfedge_iterator *hds.border_halfedges_begin()*

halfedge iterator starting with the border edges. The range [*halfedges_begin(), border_halfedges_begin()*) denotes all non-border edges. The range [*border_halfedges_begin(), halfedges_end()*) denotes all border edges.
Precondition: *normalize_border()* has been called and no halfedge insertion or removal and no change in border status of the halfedges have occurred since then.

advanced

Has Models

CGAL::HalfedgeDS_default . . . . page 1867
CGAL::HalfedgeDS_list . . . . page 1880
CGAL::HalfedgeDS_vector . . . . page 1882

See Also

HalfedgeDSItems . . . . page 1854
CGAL::Polyhedron_3<Traits> . . . . page 1795
CGAL::HalfedgeDS_vertex_base<Refs> . . . . page 1883
CGAL::HalfedgeDS_halfedge_base<Refs> . . . . page 1871
CGAL::HalfedgeDS_face_base<Refs> . . . . page 1868
CGAL::HalfedgeDS_items_decorator<HDS> . . . . page 1875
CGAL::HalfedgeDS_decorator<HDS> . . . . page 1860
CGAL::HalfedgeDS_const_decorator<HDS> . . . . page 1858

Concept

HalfedgeDSFace

Definition

The concept HalfedgeDSFace defines the requirements for the local *Face* type in the *HalfedgeDS* concept. It is also required in the *Face_wrapper<Refs,Traits>* member class template of an items class, see the *HalfedgeDSItems* concept.

A face optionally stores a reference to an incident halfedge that points to the face. A type tag indicates whether the related member functions are supported. Figure 26.3 depicts the relationship between a halfedge and its incident halfedges, vertices, and faces.

For the protection of the integrity of the data structure classes such as *CGAL::Polyhedron_3* are allowed to redefine the modifying member functions to be private. In order to make them accessible for the halfedge data structure they must be derived from a base class *Base* where the modifying member functions are still public. (The protection can be bypassed by the user, but not by accident.)

Types

HalfedgeDSFace:: HalfedgeDS	instantiated *HalfedgeDS* ($\equiv$ *Refs*).
HalfedgeDSFace:: Base	base class that allows modifications.
HalfedgeDSFace:: Vertex	model of *HalfedgeDSVertex*.
HalfedgeDSFace:: Halfedge	model of *HalfedgeDSHalfedge*.
HalfedgeDSFace:: Vertex_handle	handle to vertex.
HalfedgeDSFace:: Halfedge_handle	handle to halfedge.
HalfedgeDSFace:: Face_handle	handle to face.
HalfedgeDSFace:: Vertex_const_handle	
HalfedgeDSFace:: Halfedge_const_handle	
HalfedgeDSFace:: Face_const_handle	
HalfedgeDSFace:: Supports_face_halfedge	*CGAL::Tag_true* or *CGAL::Tag_false*.

Creation

HalfedgeDSFace f;	default constructor.

Operations available if *Supports_face_halfedge* $\equiv$ *CGAL::Tag_true*

Halfedge_handle	f*.halfedge()*	
Halfedge_const_handle	f*.halfedge() const*	incident halfedge that points to f.
void	f*.set_halfedge(Halfedge_handle h)*	sets incident halfedge to h.

Has Models

CGAL::HalfedgeDS_face_base<Refs> . . . . page 1868
CGAL::HalfedgeDS_face_min_base<Refs> . . . . page 1870

See Also

HalfedgeDS<Traits,Items,Alloc> . . . . page 1843
HalfedgeDSItems . . . . page 1854
HalfedgeDSVertex . . . . page 1856
HalfedgeDSHalfedge . . . . page 1852

HalfedgeDSHalfedge

Definition

The concept HalfedgeDSHalfedge defines the requirements for the local *Halfedge* type in the *HalfedgeDS* concept. It is also required in the *Halfedge_wrapper<Refs,Traits>* member class template of an items class, see the *HalfedgeDSItems* concept.

A halfedge is an oriented edge between two vertices. It is always paired with a halfedge pointing in the opposite direction. The *opposite()* member function returns this halfedge of opposite orientation. The *next()* member function points to the successor halfedge along the face – or if the halfedge is a border halfedge – along the open region. A halfedge optionally stores a reference to the previous halfedge along the face, a reference to an incident vertex, and a reference to an incident face. Type tags indicate whether the related member functions are supported. Figure 26.3 depicts the relationship between a halfedge and its incident halfedges, vertices, and faces.

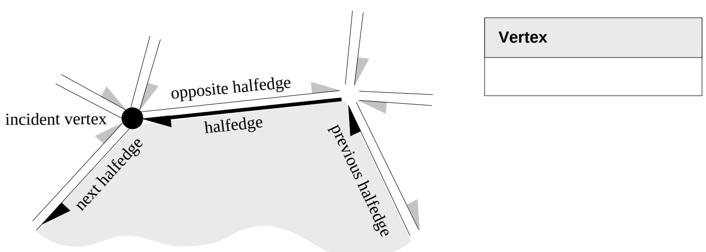


Figure 26.3: The three classes *Vertex*, *Halfedge*, and *Face* of the halfedge data structure. Member functions with shaded background are mandatory. The others are optionally supported.

For the protection of the integrity of the data structure classes such as *CGAL::Polyhedron_3* are allowed to redefine the modifying member functions to be private. In order to make them accessible for the halfedge data structure they must be derived from a base class *Base* where the modifying member functions are still public. Even more protection is provided for the *set_opposite()* member function. The base class *Base_base* provides access to it. (The protection could be bypassed also by an user, but not by accident.)

Types

HalfedgeDSHalfedge:: HalfedgeDS	instantiated *HalfedgeDS* ($\equiv$ *Refs*).
HalfedgeDSHalfedge:: Base	base class that allows modifications.
HalfedgeDSHalfedge:: Base_base	base class to access *set_opposite()*.
HalfedgeDSHalfedge:: Vertex	model of *HalfedgeDSVertex*.
HalfedgeDSHalfedge:: Face	model of *HalfedgeDSFace*.
HalfedgeDSHalfedge:: Vertex_handle	handle to vertex.
HalfedgeDSHalfedge:: Halfedge_handle	handle to halfedge.
HalfedgeDSHalfedge:: Face_handle	handle to face.
HalfedgeDSHalfedge:: Vertex_const_handle	
HalfedgeDSHalfedge:: Halfedge_const_handle	
HalfedgeDSHalfedge:: Face_const_handle	
HalfedgeDSHalfedge:: Supports_halfedge_prev	*CGAL::Tag_true* or *CGAL::Tag_false*.

HalfedgeDSHalfedge:: Supports_halfedge_vertex	~
HalfedgeDSHalfedge:: Supports_halfedge_face	~

Creation

HalfedgeDSHalfedge h;	default constructor.

Operations

Halfedge_handle	*h.opposite()*	
Halfedge_const_handle	*h.opposite() const*	the opposite halfedge.
void	*h.set_opposite(Halfedge_handle h)*	sets opposite halfedge to *h*.
Halfedge_handle	*h.next()*	
Halfedge_const_handle	*h.next() const*	the next halfedge around the face.
void	*h.set_next(Halfedge_handle h)*	sets next halfedge to *h*.
bool	*h.is_border() const*	is true if *h* is a border halfedge.

Operations available if *Supports_halfedge_prev* ≡ *CGAL::Tag_true*

Halfedge_handle	*h.prev()*	
Halfedge_const_handle	*h.prev() const*	the previous halfedge around the face.
void	*h.set_prev(Halfedge_handle h)*	sets prev halfedge to *h*.

Operations available if *Supports_halfedge_vertex* ≡ *CGAL::Tag_true*

Vertex_handle	*h.vertex()*	
Vertex_const_handle	*h.vertex() const*	the incident vertex of *h*.
void	*h.set_vertex(Vertex_handle v)*	sets incident vertex to *v*.

Operations available if *Supports_halfedge_face* ≡ *CGAL::Tag_true*

Face_handle	*h.face()*	
Face_const_handle	*h.face() const*	the incident face of *h*. If *h* is a border halfedge the result is default construction of the handle.
void	*h.set_face(Face_handle f)*	sets incident face to *f*.

Has Models

CGAL::HalfedgeDS_halfedge_base<Refs> . . . . page 1871
CGAL::HalfedgeDS_halfedge_min_base<Refs> . . . . page 1872

See Also

HalfedgeDS<Traits,Items,Alloc> . . . . page 1843
HalfedgeDSItems . . . . page 1854
HalfedgeDSVertex . . . . page 1856
HalfedgeDSFace . . . . page 1850

Concept

HalfedgeDSItems

Definition

The concept HalfedgeDSItems wraps the three item types – vertex, halfedge, and face – for a halfedge data structure. A HalfedgeDSItems contains three member class templates named *Vertex_wrapper*, *Halfedge_wrapper*, and *Face_wrapper*, each with two template parameters, *Refs* and *Traits*. *Refs* requires an instantiated halfedge data structure *HalfedgeDS* as argument, *Traits* is a geometric traits class supplied by the class that uses the halfedge data structure as internal representation. *Traits* is not used by the halfedge data structure itself. These three member class templates provide a local type named *Vertex*, *Halfedge*, and *Face* respectively. The requirements on these types are described on page 1856, page 1852, and page 1850 respectively.

Types

HalfedgeDSItems:: Vertex_wrapper<Refs,Traits>::Vertex	model of *HalfedgeDSVertex*.
HalfedgeDSItems:: Halfedge_wrapper<Refs,Traits>::Halfedge	model of *HalfedgeDSHalfedge*.
HalfedgeDSItems:: Face_wrapper<Refs,Traits>::Face	model of *HalfedgeDSFace*.

Has Models

CGAL::HalfedgeDS_min_items . . . . page 1879
CGAL::HalfedgeDS_items_2 . . . . page 1873
CGAL::Polyhedron_items_3 . . . . page 1820

See Also

HalfedgeDS<Traits,Items,Alloc> . . . . page 1843
HalfedgeDSVertex . . . . page 1856
HalfedgeDSHalfedge . . . . page 1852
HalfedgeDSFace . . . . page 1850
PolyhedronItems_3 . . . . page 1818
CGAL::HalfedgeDS_vertex_base<Refs> . . . . page 1883
CGAL::HalfedgeDS_halfedge_base<Refs> . . . . page 1871
CGAL::HalfedgeDS_face_base<Refs> . . . . page 1868

Example

The following example shows the canonical implementation of the *CGAL::HalfedgeDS_min_items* class. It uses the base classes for the item types that are provided in the library.

```
struct HalfedgeDS_min_items {
    template < class Refs, class Traits>
    struct Vertex_wrapper {
        typedef CGAL::HalfedgeDS_vertex_min_base< Refs>   Vertex;
    };
    template < class Refs, class Traits>
    struct Halfedge_wrapper {
        typedef CGAL::HalfedgeDS_halfedge_min_base< Refs> Halfedge;
```

```
    };
    template < class Refs, class Traits>
    struct Face_wrapper {
        typedef CGAL::HalfedgeDS_face_min_base< Refs>     Face;
    };
};
```

See page 1873 for an example implementation of the *CGAL::HalfedgeDS_items_2* class.

HalfedgeDSVertex

Definition

The concept HalfedgeDSVertex defines the requirements for the local *Vertex* type in the *HalfedgeDS* concept. It is also required in the *Vertex_wrapper<Refs,Traits>* member class template of an items class, see the *HalfedgeDSItems* concept.

A vertex optionally stores a reference to an incident halfedge that points to the vertex. A type tag indicates whether the related member functions are supported. Figure 26.3 depicts the relationship between a halfedge and its incident halfedges, vertices, and faces.

For the protection of the integrity of the data structure classes such as *CGAL::Polyhedron_3* are allowed to redefine the modifying member functions to be private. In order to make them accessible for the halfedge data structure they must be derived from a base class *Base* where the modifying member functions are still public. (The protection could be bypassed by an user, but not by accident.)

Types

HalfedgeDSVertex:: HalfedgeDS	instantiated *HalfedgeDS* ($\equiv$ *Refs*).
HalfedgeDSVertex:: Base	base class that allows modifications.
HalfedgeDSVertex:: Halfedge	model of *HalfedgeDSHalfedge*.
HalfedgeDSVertex:: Face	model of *HalfedgeDSFace*.
HalfedgeDSVertex:: Vertex_handle	handle to vertex.
HalfedgeDSVertex:: Halfedge_handle	handle to halfedge.
HalfedgeDSVertex:: Face_handle	handle to face.
HalfedgeDSVertex:: Vertex_const_handle	
HalfedgeDSVertex:: Halfedge_const_handle	
HalfedgeDSVertex:: Face_const_handle	
HalfedgeDSVertex:: Supports_vertex_halfedge	*CGAL::Tag_true* or *CGAL::Tag_false*.

Creation

HalfedgeDSVertex v;	default constructor.

Operations available if *Supports_vertex_halfedge* $\equiv$ *CGAL::Tag_true*

Halfedge_handle	*v.halfedge()*	
Halfedge_const_handle	*v.halfedge() const*	incident halfedge that points to *v*.
void	*v.set_halfedge(Halfedge_handle h)*	sets incident halfedge to *h*.

Has Models

CGAL::HalfedgeDS_vertex_base<Refs> . . . . page 1883
CGAL::HalfedgeDS_vertex_min_base<Refs> . . . . page 1885

See Also

HalfedgeDS<Traits,Items,Alloc> . . . . page 1843
HalfedgeDSItems . . . . page 1854
HalfedgeDSHalfedge . . . . page 1852
HalfedgeDSFace . . . . page 1850

Class

CGAL::HalfedgeDS_const_decorator<HDS>

Definition

The classes *CGAL::HalfedgeDS_items_decorator<HDS>*, *CGAL::HalfedgeDS_decorator<HDS>*, and *CGAL::HalfedgeDS_const_decorator<HDS>* provide additional functions to examine and to modify a halfedge data structure *HDS*. The class *CGAL::HalfedgeDS_items_decorator<HDS>* provides additional functions for vertices, halfedges, and faces of a halfedge data structure without knowing the containing halfedge data structure. The class *CGAL::HalfedgeDS_decorator<HDS>* stores a reference to the halfedge data structure and provides functions that modify the halfedge data structure, for example Euler-operators. The class *CGAL::HalfedgeDS_const_decorator<HDS>* stores a const reference to the halfedge data structure. It contains non-modifying functions, for example the test for validness of the data structure.

All these additional functions take care of the different capabilities a halfedge data structure may have or may not have. The functions evaluate the type tags of the halfedge data structure to decide on the actions. If a particular feature is not supported nothing is done. Note that for example the creation of new halfedges is mandatory for all halfedge data structures and will not appear here again.

#include <CGAL/HalfedgeDS_const_decorator.h>

Inherits From

CGAL::HalfedgeDS_items_decorator<HDS> . . . . page 1875

Creation

HalfedgeDS_const_decorator<HDS> D(const HDS& hds);

keeps internally a const reference to *hds*.

Validness Checks

A halfedge data structure has no definition of validness of its own, but a useful set of tests is defined with the following levels:

Level 0 The number of halfedges is even. All pointers except the vertex pointer and the face pointer for border halfedges are unequal to their respective default construction value. For all halfedges h: The opposite halfedge is different from h and the opposite of the opposite is equal to h. The next of the previous halfedge is equal to h. For all vertices v: the incident vertex of the incident halfedge of v is equal to v. The halfedges around v starting with the incident halfedge of v form a cycle. For all faces f: the incident face of the incident halfedge of f is equal to f. The halfedges around f starting with the incident halfedge of f form a cycle. Redundancies among internal variables are tested, e.g., that iterators enumerate as many items as the related size value indicates.

Level 1 All tests of level 0. For all halfedges h: The incident vertex of h exists and is equal to the incident vertex of the opposite of the next halfedge. The incident face (or hole) of h is equal to the incident face (or hole) of the next halfedge.

Level 2 All tests of level 1. The sum of all halfedges that can be reached through the vertices must be equal to the number of all halfedges, i.e., all halfedges incident to a vertex must form a single cycle.

Level 3 All tests of level 2. The sum of all halfedges that can be reached through the faces must be equal to the number of all halfedges, i.e., all halfedges surrounding a face must form a single cycle (no holes in faces).

Level 4 All tests of level 3 and *normalized_border_is_valid*.

bool

D.is_valid(bool verbose = false, int level = 0) const

returns *true* if the halfedge data structure *hds* is valid with respect to the *level* value as defined above. If *verbose* is *true*, statistics are written to *cerr*.

bool

D.normalized_border_is_valid(bool verbose = false) const

returns *true* if the border halfedges are in normalized representation, which is when enumerating all halfedges with the halfedge iterator the following holds: The non-border edges precede the border edges. For border edges, the second halfedge is a border halfedge. (The first halfedge may or may not be a border halfedge.) The halfedge iterator *border_halfedges_begin()* denotes the first border edge. If *verbose* is *true*, statistics are written to *cerr*.

See Also

CGAL::HalfedgeDS_items_decorator<HDS> . . . . page 1875
CGAL::HalfedgeDS_decorator<HDS> . . . . page 1860

Example

The following program fragment illustrates the implementation of a *is_valid()* member function for a simplified polyhedron class. We assume here that the level three check is the appropriate default for polyhedral surfaces.

```
namespace CGAL {
    template <class Traits>
    class Polyhedron {
        typedef HalfedgeDS_default<Traits> HDS;
        HDS hds;
    public:
        // ...
        bool is_valid( bool verb = false, int level = 0) const {
            Verbose_ostream verr(verb);
            verr << "begin Polyhedron::is_valid( verb=true, level = " <<
   level
                 << "):" << std::endl;
            HalfedgeDS_const_decorator<HDS> decorator(hds);
            bool valid = decorator.is_valid( verb, level + 3);
            // further checks ...
        }
    };
}
```

Class

CGAL::HalfedgeDS_decorator<HDS>

Definition

The classes *CGAL::HalfedgeDS_items_decorator<HDS>*, *CGAL::HalfedgeDS_decorator<HDS>*, and *CGAL::HalfedgeDS_const_decorator<HDS>* provide additional functions to examine and to modify a halfedge data structure *HDS*. The class *CGAL::HalfedgeDS_items_decorator<HDS>* provides additional functions for vertices, halfedges, and faces of a halfedge data structure without knowing the containing halfedge data structure. The class *CGAL::HalfedgeDS_decorator<HDS>* stores a reference to the halfedge data structure and provides functions that modify the halfedge data structure, for example Euler-operators. The class *CGAL::HalfedgeDS_const_decorator<HDS>* stores a const reference to the halfedge data structure. It contains non-modifying functions, for example the test for validness of the data structure.

All these additional functions take care of the different capabilities a halfedge data structure may have or may not have. The functions evaluate the type tags of the halfedge data structure to decide on the actions. If a particular feature is not supported nothing is done. Note that for example the creation of new halfedges is mandatory for all halfedge data structures and will not appear here again.

#include <CGAL/HalfedgeDS_decorator.h>

Inherits From

CGAL::HalfedgeDS_items_decorator<HDS> . . . . page 1875

Creation

HalfedgeDS_decorator<HDS> D(HDS& hds);	keeps internally a reference to *hds*.

Creation of New Items

Vertex_handle	*D.vertices_push_back(Vertex v)*	appends a copy of *v* to *hds* if vertices are supported. Returns a handle of the new vertex, or *Vertex_handle()* otherwise.
Face_handle	*D.faces_push_back(Face f)*	appends a copy of *f* to *hds* if faces are supported. Returns a handle of the new face, or *Face_handle()* otherwise.

Creation of New Composed Items

Halfedge_handle	*D.create_loop()*	returns handle of a halfedge from a newly created loop in *hds* consisting of a single closed edge, one vertex and two faces (if supported respectively).
Halfedge_handle	*D.create_segment()*	returns a halfedge from a newly created segment in *hds* consisting of a single open edge, two vertices and one face (if supported respectively).

Removal of Elements

The following member functions do *not* update affected incidence relations except if mentioned otherwise.

void	*D.vertices_pop_front()*	removes the first vertex if vertices are supported. *Requirement*: *Supports_removal* $\equiv$ *CGAL::Tag_true*.
void	*D.vertices_pop_back()*	removes the last vertex if vertices are supported.
void	*D.vertices_erase(Vertex_handle v)*	removes the vertex v if vertices are supported. *Requirement*: *Supports_removal* $\equiv$ *CGAL::Tag_true*.
void	*D.vertices_erase(Vertex_handle first, Vertex_handle last)*	removes the range $[first, last)$ if vertices are supported. *Requirement*: *Supports_removal* $\equiv$ *CGAL::Tag_true*.
void	*D.faces_pop_front()*	removes the first face if faces are supported. *Requirement*: *Supports_removal* $\equiv$ *CGAL::Tag_true*.
void	*D.faces_pop_back()*	removes the last face if faces are supported.
void	*D.faces_erase(Face_handle f)*	removes the face f if faces are supported. *Requirement*: *Supports_removal* $\equiv$ *CGAL::Tag_true*.
void	*D.faces_erase(Face_handle first, Face_handle last)*	removes the range $[first, last)$ if faces are supported. *Requirement*: *Supports_removal* $\equiv$ *CGAL::Tag_true*.
void	*D.erase_face(Halfedge_handle h)*	removes the face incident to h from *hds* and changes all halfedges incident to the face into border edges or removes them from the halfedge data structure if they were already border edges. If this creates isolated vertices they get removed as well. See *make_hole(h)* for a more specialized variant. *Precondition*: *h->is_border() == false*. *Requirement*: If faces are supported, *Supports_removal* $\equiv$ *CGAL::Tag_true*.
void	*D.erase_connected_component(Halfedge_handle h)*	removes the vertices, halfedges, and faces that belong to the connected component of h. *Precondition*: For all halfedges g in the connected component *g.next() != Halfedge_handle()*. *Requirement*: *Supports_removal* $\equiv$ *CGAL::Tag_true*.
unsigned int	*D.keep_largest_connected_components(unsigned int nb_components_to_keep)*	Erases the small connected components and the isolated vertices. Keep *nb_components_to_keep* largest connected components. Returns the number of connected components erased (ignoring isolated vertices). *Requirement*: supports vertices, halfedges, and removal operation.

Modifying Functions (For Border Halfedges)

void *D.make_hole(Halfedge_handle h)*

removes the face incident to *h* from *hds* and creates a hole.
Precondition: *h != Halfedge_handle()* and *!(h->is_border())*.
Requirement: If faces are supported, *Supports_removal* ≡ *CGAL::Tag_true*.

Halfedge_handle *D.fill_hole(Halfedge_handle h)*

fills the hole incident to *h* with a new face from *hds*. Returns *h*.
Precondition: *h != Halfedge_handle()* and *h->is_border()*.

Halfedge_handle *D.fill_hole(Halfedge_handle h, Face f)*

fills the hole incident to *h* with a copy of face f. Returns *h*.
Precondition: *h != Halfedge_handle()* and *h->is_border()*.

Halfedge_handle *D.add_face_to_border(Halfedge_handle h, Halfedge_handle g)*

extends the surface with a new face from *hds* into the hole incident to *h* and *g*. It creates a new edge connecting the vertex denoted by *g* with the vertex denoted by *h* and fills this separated part of the hole with a new face, such that the new face is incident to *g*. Returns the new halfedge that is incident to the new face.
Precondition: *h != Halfedge_handle()*, *g != Halfedge_handle()*, *h->is_border()*, *g->is_border()* and *g* can be reached along the hole starting with *h*.

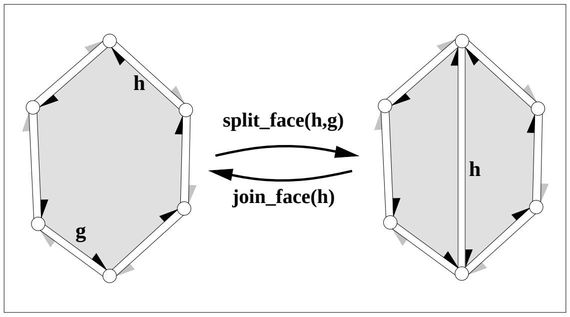
Halfedge_handle *D.add_face_to_border(Halfedge_handle h, Halfedge_handle g, Face f)*

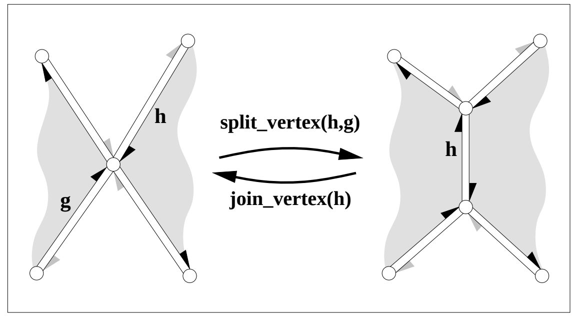
extends the surface with a copy of face f into the hole incident to *h* and *g*. It creates a new edge connecting the tip of *g* with the tip of *h* and fills this separated part of the hole with a copy of face f, such that the new face is incident to *g*. Returns the new halfedge that is incident to the new face.
Precondition: *h != Halfedge_handle()*, *g != Halfedge_handle()*, *h->is_border()*, *g->is_border()* and *g* can be reached along the hole starting with *h*.

Modifying Functions (Euler Operators)

The following Euler operations modify consistently the combinatorial structure of the halfedge data structure. The geometry remains unchanged. Note that well known graph operations are also captured with these Euler operators, for example an edge contraction is equal to a *join_vertex()* operation, or an edge removal to *join_face()*.

Given a halfedge data structure *hds* and a halfedge handle *h* four special applications of the Euler operators are worth mentioning: *split_vertex(h,h)* results in an antenna emanating from the tip of *h*; *split_vertex(h,h->next()->opposite())* results in an edge split of the halfedge *h->next* with a new vertex in-between; *split_face(h,h)* results in a loop directly following *h*; and *split_face(h,h->next())* results in a bridge parallel to the halfedge *h->next* with a new face in-between.





Halfedge_handle   *D.split_face(Halfedge_handle h, Halfedge_handle g)*

splits the face incident to *h* and *g* into two faces with a new diagonal between the two vertices denoted by *h* and *g* respectively. The second (new) face obtained from *hds* is a copy of the first face. Returns *h->next()* after the operation, i.e., the new diagonal. The new face is to the right of the new diagonal, the old face is to the left. The time is proportional to the distance from *h* to *g* around the face.

Halfedge_handle   *D.join_face(Halfedge_handle h)*

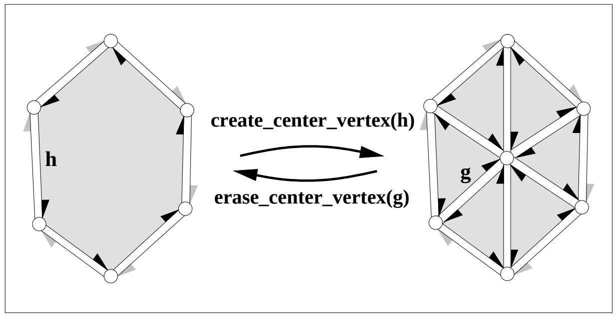
joins the two faces incident to *h*. The face incident to *h->opposite()* gets removed from *hds*. Both faces might be holes. Returns the predecessor of *h* around the face. The invariant *join_face(split_face(h, g))* returns *h* and keeps the data structure unchanged. The time is proportional to the size of the face removed and the time to compute *h->prev()*.
Requirement: *Supports_removal* $\equiv$ *CGAL::Tag_true*.

Halfedge_handle   *D.split_vertex(Halfedge_handle h, Halfedge_handle g)*

splits the vertex incident to *h* and *g* into two vertices and connects them with a new edge. The second (new) vertex obtained from *hds* is a copy of the first vertex. Returns *h->next()->opposite()* after the operation, i.e., the new edge in the orientation towards the new vertex. The time is proportional to the distance from *h* to *g* around the vertex.

Halfedge_handle   *D.join_vertex(Halfedge_handle h)*

joins the two vertices incident to *h*. The vertex denoted by *h->opposite()* gets removed by *hds*. Returns the predecessor of *h* around the vertex, i.e., *h->opposite()->prev()*. The invariant *join_vertex(split_vertex(h, g))* returns *h* and keeps the polyhedron unchanged. The time is proportional to the degree of the vertex removed and the time to compute *h->prev()* and *h->opposite()->prev()*.
Requirement: *Supports_removal* $\equiv$ *CGAL::Tag_true*.

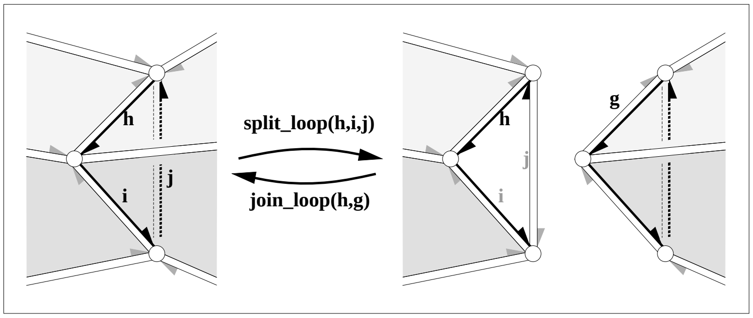


Halfedge_handle *D.create_center_vertex(Halfedge_handle h)*

barycentric triangulation of *h->face()*. Creates a new vertex, a copy of *h->vertex()*, and connects it to each vertex incident to *h->face()* splitting *h->face()* into triangles. *h* remains incident to the original face, all other triangles are copies of this face. Returns the halfedge *h->next()* after the operation, i.e., a halfedge pointing to the new vertex. The time is proportional to the size of the face.
Precondition: *h* is not a border halfedge.

Halfedge_handle *D.erase_center_vertex(Halfedge_handle g)*

reverses *create_center_vertex*. Erases the vertex pointed to by *g* and all incident halfedges thereby merging all incident faces. Only *g->face()* remains. The neighborhood of *g->vertex()* may not be triangulated, it can have larger faces. Returns the halfedge *g->prev()*. Thus, the invariant *h* == *erase_center_vertex(create_center_vertex(h))* holds if *h* is not a border halfedge. The time is proportional to the sum of the size of all incident faces.
Precondition: None of the incident faces of *g->vertex()* is a hole. There are at least two distinct faces incident to the faces that are incident to *g->vertex()*. (This prevents the operation from collapsing a volume into two faces glued together with opposite orientations, such as would happen with any vertex of a tetrahedron.)
Requirement: *Supports_removal* $\equiv$ *CGAL::Tag_true*.



Halfedge_handle *D.split_loop(Halfedge_handle h, Halfedge_handle i, Halfedge_handle j)*

cuts the halfedge data structure into two parts along the cycle (h,i,j). Three new vertices (one copy for each vertex in the cycle) and three new halfedges (one copy for each halfedge in the cycle), and two new triangles are created. h,i,j will be incident to the first new triangle. The return value will be the halfedge incident to the second new triangle which is the copy of *h-opposite()*.
Precondition: h,i,j denote distinct, consecutive vertices of the halfedge data structure and form a cycle: i.e., *h->vertex() == i->opposite()->vertex()*, ..., *j->vertex() == h->opposite()->vertex()*.

Halfedge_handle *D.join_loop(Halfedge_handle h, Halfedge_handle g)*

glues the boundary of the two faces denoted by *h* and *g* together and returns *h*. Both faces and the vertices along the face denoted by *g* gets removed. Both faces may be holes. The invariant *join_loop(h, split_loop(h, i, j))* returns *h* and keeps the data structure unchanged.
Precondition: The faces denoted by *h* and *g* are different and have equal degree (i.e., number of edges).
Requirement: *Supports_removal* $\equiv$ *CGAL::Tag_true*.

Validness Checks

These operations are the same as for *CGAL::HalfedgeDS_const_decorator<HDS>*. See their documentation on page 1858.

bool *D.is_valid(bool verbose = false, int level = 0) const*

bool *D.normalized_border_is_valid(bool verbose = false) const*

Miscellaneous

void *D.inside_out()* reverses face orientations.
Precondition: *is_valid()* of level three.

See Also

CGAL::HalfedgeDS_items_decorator<HDS> .... page 1875
CGAL::HalfedgeDS_const_decorator<HDS> .... page 1858

Example

The following program fragment illustrates the implementation of the Euler operator *split_vertex()* for a simplified polyhedron class.

```
template <class Traits>
namespace CGAL {
    class Polyhedron {
        typedef HalfedgeDS_default<Traits> HDS;
        HDS hds;
    public:
        // ...
        Halfedge_handle split_vertex( Halfedge_handle h, Halfedge_handle
    g) {
            HalfedgeDS_decorator<HDS> D(hds);
            // Stricter preconditions than for HalfedgeDS only.
            CGAL_precondition( D.get_vertex(h) == D.get_vertex(g));
            CGAL_precondition( h != g);
            return D.split_vertex( h, g);
        }
    };
}
```

CGAL::HalfedgeDS_default<Traits,HalfedgeDSItems,Alloc>

Definition

template < class Traits,
class HalfedgeDSItems = CGAL::HalfedgeDS_items_2,
class Alloc = CGAL_ALLOCATOR(int)>
class HalfedgeDS_default;

The class *HalfedgeDS_default<Traits,HalfedgeDSItems,Alloc>* is a model for the *HalfedgeDS* concept. The second template parameter *HalfedgeDSItems* has a default argument *CGAL::HalfedgeDS_items_2*. The third template parameter *Alloc* uses the CGAL default allocator as default setting. *HalfedgeDS_default< Traits,HalfedgeDSItems,Alloc>* is a list-based representation with bidirectional iterators that supports removal.

#include <CGAL/HalfedgeDS_default.h>

Is Model for the Concepts

HalfedgeDS<Traits,Items,Alloc> . . . . page 1843

Types

typedef bidirectional_iterator_tag	*iterator_category;*
typedef CGAL::Tag_true	*Supports_removal;*

See Also

CGAL::HalfedgeDS_list . . . . page 1880
CGAL::HalfedgeDS_vector . . . . page 1882
HalfedgeDSItems . . . . page 1854
CGAL::HalfedgeDS_items_2 . . . . page 1873
CGAL::Polyhedron_3<Traits> . . . . page 1795
CGAL::HalfedgeDS_items_decorator<HDS> . . . . page 1875
CGAL::HalfedgeDS_decorator<HDS> . . . . page 1860
CGAL::HalfedgeDS_const_decorator<HDS> . . . . page 1858

Implementation

Currently, *HalfedgeDS_default<Traits,HalfedgeDSItems,Alloc>* is derived from *CGAL::HalfedgeDS_list< Traits>*. The copy constructor and the assignment operator need $O(n)$ time with n the total number of vertices, halfedges, and faces. The former suboptimal implementation with an $O(n \log n)$ runtime has been replaced with a faster implementation based on hashing for the pointer lookup.

Class

CGAL::HalfedgeDS_face_base<Refs>

Definition

The class *HalfedgeDS_face_base<Refs>* is a model of the *HalfedgeDSFace* concept. *Refs* is an instantiation of a *HalfedgeDS*. The template declaration of *HalfedgeDS_face_base<Refs>* has three parameters with some defaults that allow to select various flavors of faces. The declaration is best explained with the two following declarations, essentially hiding an implementation dependent default setting:

template <class Refs, class T = CGAL::Tag_true>
class HalfedgeDS_face_base;

template <class Refs, class T, class Plane>
class HalfedgeDS_face_base;

HalfedgeDS_face_base<Refs> defines a face including a reference to an incident halfedge.

CGAL::HalfedgeDS_face_base<Refs,CGAL::Tag_false> is a face without a reference to an incident halfedge. It is empty besides the required type definitions. It can be used for deriving own faces. See also *CGAL::HalfedgeDS_face_min_base<Refs>*.

CGAL::HalfedgeDS_face_base<Refs,CGAL::Tag_true,Plane> is a face with a reference to an incident halfedge and it stores a plane equation of type *Plane*. It can be used as a face for a model of the *PolyhedronItems_3* concept.

CGAL::HalfedgeDS_face_base<Refs,CGAL::Tag_false,Plane> is a face without a reference to an incident halfedge and it stores a plane equation of type *Plane*. It can be used as a face for a model of the *PolyhedronItems_3* concept.

#include <CGAL/HalfedgeDS_face_base.h>

Is Model for the Concepts

HalfedgeDSFace . . . . page 1850

Types

HalfedgeDS_face_base<Refs>:: Plane	plane type for three argument version.

Creation

HalfedgeDS_face_base<Refs> f;	default constructor.
HalfedgeDS_face_base<Refs> f(Plane pln);	initialized with plane *pln*.

Operations

Plane&	*f.plane()*
const Plane&	*f.plane() const*

See Also

HalfedgeDS<Traits,Items,Alloc> . . . page 1843
HalfedgeDSItems . . . page 1854
PolyhedronItems_3 . . . page 1818
CGAL::HalfedgeDS_items_2 . . . page 1873
CGAL::HalfedgeDS_vertex_base<Refs> . . . page 1883
CGAL::HalfedgeDS_halfedge_base<Refs> . . . page 1871
CGAL::HalfedgeDS_face_min_base<Refs> . . . page 1870

CGAL::HalfedgeDS_face_min_base<Refs>

The class *HalfedgeDS_face_min_base<Refs>* is a model of the *HalfedgeDSFace* concept. *Refs* is an instantiation of a *HalfedgeDS*. It is equivalent to *CGAL::HalfedgeDS_face_base< Refs, CGAL::Tag_false>*. It is empty besides the required type definitions. It can be used for deriving own faces.

#include <CGAL/HalfedgeDS_face_min_base.h>

Is Model for the Concepts

HalfedgeDSFace . . . . page 1850

Creation

HalfedgeDS_face_min_base<Refs> f;	default constructor.

See Also

HalfedgeDS<Traits,Items,Alloc> . . . . page 1843
HalfedgeDSItems . . . . page 1854
PolyhedronItems_3 . . . . page 1818
CGAL::HalfedgeDS_min_items . . . . page 1879
CGAL::HalfedgeDS_vertex_min_base<Refs> . . . . page 1885
CGAL::HalfedgeDS_halfedge_min_base<Refs> . . . . page 1872
CGAL::HalfedgeDS_face_base<Refs> . . . . page 1868

CGAL::HalfedgeDS_halfedge_base<Refs>

Class

Definition

The class *HalfedgeDS_halfedge_base<Refs>* is a model of the *HalfedgeDSHalfedge* concept. *Refs* is an instantiation of a *HalfedgeDS*. The full declaration states four template parameters:

```
template < class Refs,
           class Tag_prev   = CGAL::Tag_true,
           class Tag_vertex = CGAL::Tag_true,
           class Tag_face   = CGAL::Tag_true>
class HalfedgeDS_halfedge_base;
```

If *Tag_prev* ≡ *CGAL::Tag_true* a reference to the previous halfedge is supported.

If *Tag_vertex* ≡ *CGAL::Tag_true* an incident vertex is supported.

If *Tag_face* ≡ *CGAL::Tag_true* an incident face is supported.

In all cases, a reference to the next halfedge and to the opposite halfedge is supported.

#include <CGAL/HalfedgeDS_halfedge_base.h>

Is Model for the Concepts

HalfedgeDSHalfedge . . . . page 1852

Creation

HalfedgeDS_halfedge_base<Refs> h; default constructor.

See Also

HalfedgeDS<Traits,Items,Alloc> . . . . page 1843
HalfedgeDSItems . . . . page 1854
PolyhedronItems_3 . . . . page 1818
CGAL::HalfedgeDS_items_2 . . . . page 1873
CGAL::HalfedgeDS_vertex_base<Refs> . . . . page 1883
CGAL::HalfedgeDS_face_base<Refs> . . . . page 1868
CGAL::HalfedgeDS_halfedge_min_base<Refs> . . . . page 1872

Class

CGAL::HalfedgeDS_halfedge_min_base<Refs>

Definition

The class *HalfedgeDS_halfedge_min_base<Refs>* is a model of the *HalfedgeDSHalfedge* concept. *Refs* is an instantiation of a *HalfedgeDS*. It is equivalent to *CGAL::HalfedgeDS_halfedge_base< Refs, CGAL::Tag_false, CGAL::Tag_false, CGAL::Tag_false>*. The class contains support for the next and the opposite pointer and the required type definitions. It can be used for deriving own halfedges.

#include <CGAL/HalfedgeDS_halfedge_min_base.h>

Is Model for the Concepts

HalfedgeDSHalfedge . . . . page 1852

Creation

HalfedgeDS_halfedge_min_base<Refs> h;   default constructor.

See Also

HalfedgeDS<Traits,Items,Alloc> . . . . page 1843
HalfedgeDSItems . . . . page 1854
PolyhedronItems_3 . . . . page 1818
CGAL::HalfedgeDS_min_items . . . . page 1879
CGAL::HalfedgeDS_vertex_min_base<Refs> . . . . page 1885
CGAL::HalfedgeDS_face_min_base<Refs> . . . . page 1870
CGAL::HalfedgeDS_halfedge_base<Refs> . . . . page 1871

CGAL::HalfedgeDS_items_2

Class

Definition

The class *HalfedgeDS_items_2* is a model of the *HalfedgeDSItems* concept. It uses the default types for vertices, halfedges, and faces that declare all incidences supported by a *HalfedgeDS*. The vertex also contains a point of type *Traits::Point_2*, where *Traits* is the template argument of the corresponding *HalfedgeDS*.

#include <CGAL/HalfedgeDS_items_2.h>

Is Model for the Concepts

HalfedgeDSItems . . . . page 1854

See Also

CGAL::HalfedgeDS_min_items . . . . page 1879
CGAL::Polyhedron_items_3 . . . . page 1820
HalfedgeDS<Traits,Items,Alloc> . . . . page 1843
PolyhedronItems_3 . . . . page 1818
CGAL::HalfedgeDS_vertex_base<Refs> . . . . page 1883
CGAL::HalfedgeDS_halfedge_base<Refs> . . . . page 1871
CGAL::HalfedgeDS_face_base<Refs> . . . . page 1868

Example

The following example shows the canonical implementation of the *HalfedgeDS_items_2* class. It uses the base classes for the item types that are provided in the library.

```
struct HalfedgeDS_items_2 {
    template < class Refs, class Traits>
    struct Vertex_wrapper {
        typedef typename Traits::Point_2 Point;
        typedef CGAL::HalfedgeDS_vertex_base< Refs, Tag_true, Point>
   Vertex;
    };
    template < class Refs, class Traits>
    struct Halfedge_wrapper {
        typedef CGAL::HalfedgeDS_halfedge_base< Refs> Halfedge;
    };
    template < class Refs, class Traits>
    struct Face_wrapper {
        typedef CGAL::HalfedgeDS_face_base< Refs> Face;
    };
};
```

The following example shows a class definition for a new items class derived from the *HalfedgeDS_items_2* class. It replaces the *Face_wrapper* with a new definition of a face that contains a member variable for color. The new face makes use of the face base class provided in the library.

```
// A face type with a color member variable.
template <class Refs>
struct My_face : public CGAL::HalfedgeDS_face_base<Refs> {
    CGAL::Color color;
    My_face() {}
    My_face( CGAL::Color c) : color(c) {}
};

// An items type using my face.
struct My_items : public CGAL::HalfedgeDS_items_2 {
    template <class Refs, class Traits>
    struct Face_wrapper {
        typedef My_face<Refs> Face;
    };
};
```

CGAL::HalfedgeDS_items_decorator<HDS>

Class

Definition

The classes *CGAL::HalfedgeDS_items_decorator<HDS>*, *CGAL::HalfedgeDS_decorator<HDS>*, and *CGAL::HalfedgeDS_const_decorator<HDS>* provide additional functions to examine and to modify a halfedge data structure *HDS*. The class *CGAL::HalfedgeDS_items_decorator<HDS>* provides additional functions for vertices, halfedges, and faces of a halfedge data structure without knowing the containing halfedge data structure. The class *CGAL::HalfedgeDS_decorator<HDS>* stores a reference to the halfedge data structure and provides functions that modify the halfedge data structure, for example Euler-operators. The class *CGAL::HalfedgeDS_const_decorator<HDS>* stores a const reference to the halfedge data structure. It contains non-modifying functions, for example the test for validness of the data structure.

All these additional functions take care of the different capabilities a halfedge data structure may have or may not have. The functions evaluate the type tags of the halfedge data structure to decide on the actions. If a particular feature is not supported nothing is done. Note that for example the creation of new halfedges is mandatory for all halfedge data structures and will not appear here again.

#include <CGAL/HalfedgeDS_items_decorator.h>

Types

HalfedgeDS_items_decorator<HDS>:: HalfedgeDS	halfedge data structure.
HalfedgeDS_items_decorator<HDS>:: Traits	traits class.
HalfedgeDS_items_decorator<HDS>:: Vertex	vertex type of *HalfedgeDS*.
HalfedgeDS_items_decorator<HDS>:: Halfedge	halfedge type of *HalfedgeDS*.
HalfedgeDS_items_decorator<HDS>:: Face	face type of *HalfedgeDS*.

HalfedgeDS_items_decorator<HDS>:: Vertex_handle
HalfedgeDS_items_decorator<HDS>:: Halfedge_handle
HalfedgeDS_items_decorator<HDS>:: Face_handle
HalfedgeDS_items_decorator<HDS>:: Vertex_iterator
HalfedgeDS_items_decorator<HDS>:: Halfedge_iterator
HalfedgeDS_items_decorator<HDS>:: Face_iterator

The respective *const_handle*'s and *const_iterator*'s are available as well.

HalfedgeDS_items_decorator<HDS>:: size_type
HalfedgeDS_items_decorator<HDS>:: difference_type
HalfedgeDS_items_decorator<HDS>:: iterator_category

HalfedgeDS_items_decorator<HDS>:: Supports_vertex_halfedge
HalfedgeDS_items_decorator<HDS>:: Supports_halfedge_prev
HalfedgeDS_items_decorator<HDS>:: Supports_halfedge_vertex
HalfedgeDS_items_decorator<HDS>:: Supports_halfedge_face
HalfedgeDS_items_decorator<HDS>:: Supports_face_halfedge
HalfedgeDS_items_decorator<HDS>:: Supports_removal

Creation

HalfedgeDS_items_decorator<HDS> D;	default constructor.

Access Functions

Halfedge_handle	*D.get_vertex_halfedge(Vertex_handle v)*	returns the incident halfedge of *v* if supported, *Halfedge_handle()* otherwise.
Vertex_handle	*D.get_vertex(Halfedge_handle h)*	returns the incident vertex of *h* if supported, *Vertex_handle()* otherwise.
Halfedge_handle	*D.get_prev(Halfedge_handle h)*	returns the previous halfedge of *h* if supported, *Halfedge_handle()* otherwise.
Halfedge_handle	*D.find_prev(Halfedge_handle h)*	returns the previous halfedge of *h*. Uses the *prev()* method if supported or performs a search around the face using *next()*.
Halfedge_handle	*D.find_prev_around_vertex(Halfedge_handle h)*	returns the previous halfedge of *h*. Uses the *prev()* method if supported or performs a search around the vertex using *next()*.
Face_handle	*D.get_face(Halfedge_handle h)*	returns the incident face of *h* if supported, *Face_handle()* otherwise.
Halfedge_handle	*D.get_face_halfedge(Face_handle f)*	returns the incident halfedge of *f* if supported, *Halfedge_handle()* otherwise.

Corresponding member functions for *const_handle*'s are provided as well.

Modifying Functions (Composed)

void	*D.close_tip(Halfedge_handle h) const*	makes *h->opposite()* the successor of *h*.
void	*D.close_tip(Halfedge_handle h, Vertex_handle v) const*	makes *h->opposite()* the successor of *h* and sets the incident vertex of *h* to *v*.
void	*D.insert_tip(Halfedge_handle h, Halfedge_handle v) const*	inserts the tip of the edge *h* into the halfedges around the vertex pointed to by *v*. Halfedge *h->opposite()* is the new successor of *v* and *h->next()* will be set to *v->next()*. The vertex of *h* will be set to the vertex *v* refers to if vertices are supported.
void	*D.remove_tip(Halfedge_handle h) const*	removes the edge *h->next()->opposite()* from the halfedge circle around the vertex referred to by *h*. The new successor halfedge of *h* will be *h->next()->opposite()->next()*.

void *D.insert_halfedge(Halfedge_handle h, Halfedge_handle f) const*

inserts the halfedge *h* between *f* and *f->next()*. The face of *h* will be the one *f* refers to if faces are supported.

void *D.remove_halfedge(Halfedge_handle h) const*

removes edge *h->next()* from the halfedge circle around the face referred to by *h*. The new successor of *h* will be *h->next()->next()*.

void *D.set_vertex_in_vertex_loop(Halfedge_handle h, Vertex_handle v) const*

loops around the vertex incident to *h* and sets all vertex pointers to *v*.
Precondition: *h != Halfedge_handle()*.

void *D.set_face_in_face_loop(Halfedge_handle h, Face_handle f) const*

loops around the face incident to *h* and sets all face pointers to *f*.
Precondition: *h != Halfedge_handle()*.

Halfedge_handle *D.flip_edge(Halfedge_handle h) const*

performs an edge flip. It returns *h* after rotating the edge *h* one vertex in the direction of the face orientation.
Precondition: *h != Halfedge_handle()* and both incident faces of *h* are triangles.

Modifying Functions (Primitives)

void *D.set_vertex_halfedge(Vertex_handle v, Halfedge_handle g) const*

sets the incident halfedge of *v* to *g*.

void *D.set_vertex_halfedge(Halfedge_handle h) const*

sets the incident halfedge of the vertex incident to *h* to *h*.

void *D.set_vertex(Halfedge_handle h, Vertex_handle v) const*

sets the incident vertex of *h* to *v*.

void *D.set_prev(Halfedge_handle h, Halfedge_handle g) const*

sets the previous link of *h* to *g*.

void *D.set_face(Halfedge_handle h, Face_handle f) const*

sets the incident face of *h* to *f*.

void *D.set_face_halfedge(Face_handle f, Halfedge_handle g) const*

sets the incident halfedge of *f* to *g*.

void *D.set_face_halfedge(Halfedge_handle h) const*

sets the incident halfedge of the face incident to *h* to *h*.

See Also

Example

The following program fragment illustrates how a refined halfedge class for a polyhedron can make use of the *find_prev()* member function to implement a *prev()* member function that works regardless of whether the halfedge data structure *HDS* provides a *prev()* member function for its halfedges or not. In the case that not, the implementation given here runs in time proportional to the size of the incident face. For const-correctness a second implementation with signature *Halfedge_const_handle prev() const;* is needed.

Note also the use of the static member function *halfedge_handle()* of the halfedge data structure. It converts a pointer to the halfedge into a halfedge handle. This conversion encapsulates possible adjustments for hidden data members in the true halfedge type, such as linked-list pointers.

```
struct Polyhedron_halfedge {
    // ...
    Halfedge_handle  prev() {
        CGAL::HalfedgeDS_items_decorator<HDS> decorator;
        return decorator.find_prev( HDS::halfedge_handle(this));
    }
};
```

CGAL::HalfedgeDS_min_items

Class

Definition

The class *HalfedgeDS_min_items* is a model of the *HalfedgeDSItems* concept. It defines types for vertices, halfedges, and faces that declare the minimal required incidences for a *HalfedgeDS*, which are the *next()* and the *opposite()* member function for halfedges.

#include <CGAL/HalfedgeDS_min_items.h>

Is Model for the Concepts

HalfedgeDSItems . . . . page 1854

See Also

CGAL::HalfedgeDS_items_2 . . . . page 1873
CGAL::Polyhedron_items_3 . . . . page 1820
HalfedgeDS<Traits,Items,Alloc> . . . . page 1843
PolyhedronItems_3 . . . . page 1818
CGAL::HalfedgeDS_vertex_min_base<Refs> . . . . page 1885
CGAL::HalfedgeDS_halfedge_min_base<Refs> . . . . page 1872
CGAL::HalfedgeDS_face_min_base<Refs> . . . . page 1870

Example

The following example shows the canonical implementation of the *CGAL::HalfedgeDS_min_items* class. It uses the base classes for the item types that are provided in the library.

```
struct HalfedgeDS_min_items {
    template < class Refs, class Traits>
    struct Vertex_wrapper {
        typedef CGAL::HalfedgeDS_vertex_min_base< Refs>   Vertex;
    };
    template < class Refs, class Traits>
    struct Halfedge_wrapper {
        typedef CGAL::HalfedgeDS_halfedge_min_base< Refs> Halfedge;
    };
    template < class Refs, class Traits>
    struct Face_wrapper {
        typedef CGAL::HalfedgeDS_face_min_base< Refs>     Face;
    };
};
```

Class

CGAL::HalfedgeDS_list<Traits,HalfedgeDSItems,Alloc>

Definition

The class *HalfedgeDS_list<Traits,HalfedgeDSItems,Alloc>* is a model for the *HalfedgeDS* concept. *HalfedgeDS_list<Traits,HalfedgeDSItems,Alloc>* is a list-based representation with bidirectional iterators that supports removal.

#include <CGAL/HalfedgeDS_list.h>

Is Model for the Concepts

HalfedgeDS<Traits,Items,Alloc> . . . . page 1843

Types

typedef bidirectional_iterator_tag	*iterator_category;*
typedef CGAL::Tag_true	*Supports_removal;*

Operations

Besides operations required from the concept *HalfedgeDS<Traits,Items,Alloc>*, this class supports additionally:

void *hds.vertices_splice(Vertex_iterator target, Self &source, Vertex_iterator first, Vertex_iterator last)*

inserts elements in the range [*first, last*) before position *target* and removes the elements from *source*. It takes constant time if &*source* == &*hds*; otherwise, it takes linear time in the size of the range.
Precondition: [*first, last*) is a valid range in *source*. *target* is not in the range [*first, last*).

void *hds.halfedges_splice(Halfedge_iterator target,*
Self &source,
Halfedge_iterator first,
Halfedge_iterator last)

inserts elements in the range [*first, last*) before position *target* and removes the elements from *source*. It takes constant time if &*source* == &*hds*; otherwise, it takes linear time in the size of the range.
Precondition: [*first, last*) is a valid range in *source*. *target* is not in the range [*first, last*).

void *hds.faces_splice(Face_iterator target, Self &source, Face_iterator first, Face_iterator last)*

inserts elements in the range [*first, last*) before position *target* and removes the elements from *source*. It takes constant time if &*source* == &*hds*; otherwise, it takes linear time in the size of the range.
Precondition: [*first, last*) is a valid range in *source*. *target* is not in the range [*first, last*).

See Also

CGAL::HalfedgeDS_default . . . . page 1867
CGAL::HalfedgeDS_vector . . . . page 1882
HalfedgeDSItems . . . . page 1854
CGAL::Polyhedron_3<Traits> . . . . page 1795
CGAL::HalfedgeDS_items_decorator<HDS> . . . . page 1875
CGAL::HalfedgeDS_decorator<HDS> . . . . page 1860
CGAL::HalfedgeDS_const_decorator<HDS> . . . . page 1858

Implementation

HalfedgeDS_list<Traits,HalfedgeDSItems,Alloc> uses internally the *CGAL::In_place_list* container class. The copy constructor and the assignment operator need $O(n)$ time with n the total number of vertices, halfedges, and faces. The former suboptimal implementation with an $O(n \log n)$ runtime has been replaced with a faster implementation based on hashing for the pointer lookup.

CGAL_ALLOCATOR(int) is used as default argument for the *Alloc* template parameter.

Class

CGAL::HalfedgeDS_vector<Traits,HalfedgeDSItems,Alloc>

Definition

The class *HalfedgeDS_vector<Traits,HalfedgeDSItems,Alloc>* is a model for the *HalfedgeDS* concept. *HalfedgeDS_vector<Traits,HalfedgeDSItems,Alloc>* is a vector-based representation with random access iterators that does not support removal.

#include <CGAL/HalfedgeDS_vector.h>

Is Model for the Concepts

HalfedgeDS<Traits,Items,Alloc> . . . . page 1843

Types

typedef random_access_iterator_tag	*iterator_category;*
typedef CGAL::Tag_false	*Supports_removal;*

See Also

CGAL::HalfedgeDS_default . . . . page 1867
CGAL::HalfedgeDS_list . . . . page 1880
HalfedgeDSItems . . . . page 1854
CGAL::Polyhedron_3<Traits> . . . . page 1795
CGAL::HalfedgeDS_items_decorator<HDS> . . . . page 1875
CGAL::HalfedgeDS_decorator<HDS> . . . . page 1860
CGAL::HalfedgeDS_const_decorator<HDS> . . . . page 1858

Implementation

HalfedgeDS_vector<Traits,HalfedgeDSItems,Alloc> uses internally the STL *std::vector* container class. We require that we can create a *std::vector::iterator* from a pointer. If this will not be true any longer for any major STL distribution we might switch to an internal implementation of a vector.

The capacity is restricted to the reserved size. Allocations are not possible beyond the capacity without calling reserve again. All handles and iterators are invalidated upon a reserve call that increases the capacity.

CGAL_ALLOCATOR(int) is used as default argument for the *Alloc* template parameter.

CGAL::HalfedgeDS_vertex_base<Refs>

Class

Definition

The class *HalfedgeDS_vertex_base<Refs>* is a model of the *HalfedgeDSVertex* concept. *Refs* is an instantiation of a *HalfedgeDS*. The template declaration of *HalfedgeDS_vertex_base<Refs>* has three parameters with some defaults that allow to select various flavors of vertices. The declaration is best explained with the two following declarations, essentially hiding an implementation dependent default setting:

template <class Refs, class T = CGAL::Tag_true>
class HalfedgeDS_vertex_base;

template <class Refs, class T, class Point>
class HalfedgeDS_vertex_base;

HalfedgeDS_vertex_base<Refs> defines a vertex including a reference to an incident halfedge.

CGAL::HalfedgeDS_vertex_base<Refs,CGAL::Tag_false> is a vertex without a reference to an incident halfedge. It is empty besides the required type definitions. It can be used for deriving own vertex implementations. See also *CGAL::HalfedgeDS_vertex_min_base<Refs>*.

CGAL::HalfedgeDS_vertex_base<Refs,CGAL::Tag_true,Point> is a vertex with a reference to an incident halfedge and it stores a point of type *Point*. It can be used as a vertex for a model of the *PolyhedronItems_3* concept.

CGAL::HalfedgeDS_vertex_base<Refs,CGAL::Tag_false,Point> is a vertex without a reference to an incident halfedge and it stores a point of type *Point*. It can be used as a vertex for a model of the *PolyhedronItems_3* concept.

#include <CGAL/HalfedgeDS_vertex_base.h>

Is Model for the Concepts

HalfedgeDSVertex . . . . page 1856

Types

HalfedgeDS_vertex_base<Refs>:: Point	point type for three argument version.

Creation

HalfedgeDS_vertex_base<Refs> v;	default constructor.
HalfedgeDS_vertex_base<Refs> v(Point p);	initialized with point *p*.

Operations

Point&	*v.point()*
const Point&	*v.point() const*

See Also

HalfedgeDS<Traits,Items,Alloc> . . . . page 1843
HalfedgeDSItems . . . . page 1854
PolyhedronItems_3 . . . . page 1818
CGAL::HalfedgeDS_items_2 . . . . page 1873
CGAL::HalfedgeDS_halfedge_base<Refs> . . . . page 1871
CGAL::HalfedgeDS_face_base<Refs> . . . . page 1868
CGAL::HalfedgeDS_vertex_min_base<Refs> . . . . page 1885

CGAL::HalfedgeDS_vertex_min_base<Refs>

Class

Definition

The class *HalfedgeDS_vertex_min_base<Refs>* is a model of the *HalfedgeDSVertex* concept. *Refs* is an instantiation of a *HalfedgeDS*. It is equivalent to *CGAL::HalfedgeDS_vertex_base< Refs, CGAL::Tag_false>*. It is empty besides the required type definitions. It can be used for deriving own vertices.

#include <CGAL/HalfedgeDS_vertex_min_base.h>

Is Model for the Concepts

HalfedgeDSVertex . . . . page 1856

Creation

HalfedgeDS_vertex_min_base<Refs> v;	default constructor.

See Also

HalfedgeDS<Traits,Items,Alloc> . . . . page 1843
HalfedgeDSItems . . . . page 1854
PolyhedronItems_3 . . . . page 1818
CGAL::HalfedgeDS_min_items . . . . page 1879
CGAL::HalfedgeDS_halfedge_min_base<Refs> . . . . page 1872
CGAL::HalfedgeDS_face_min_base<Refs> . . . . page 1870
CGAL::HalfedgeDS_vertex_base<Refs> . . . . page 1883

Chapter 27

Combinatorial Maps

Guillaume Damiand

Contents

27.1 Introduction

A *d*-dimensional combinatorial map is a data structure representing an orientable subdivided *d*-dimensional object obtained by taking *d*D cells, and allowing to glue *d*D cells along *(d-1)*D cells. It provides a description of all the cells of the subdivision (for example vertices and edges), together with incidence and adjacency relationships. This package is a generalization of the halfedge data structure to higher dimension.[1]

We denote *i*-cell for an *i*-dimensional cell (for example in 3D, 0-cells are *vertices*, 1-cells are *edges*, 2-cells are *facets*, and 3-cells are *volumes*). A *boundary relation* is defined on these cells, giving for each *i*-cell *c* the set of *(i-1)*-cells contained in the boundary of *c*. Two cells *c1* and *c2* are *incident* if there is a path of cells, starting from the cell of biggest dimension to the other cell, such that each cell of the path (except the first one) belongs to the boundary of the previous cell in the path. Two *i*-cells *c3* and *c4* are *adjacent* if there is an *(i-1)*-cell incident to both *c3* and *c4*. You can see an example of a 2D object and a 3D object in Figure 27.1 showing some cells of the subdivision and some adjacency and incidence relations.

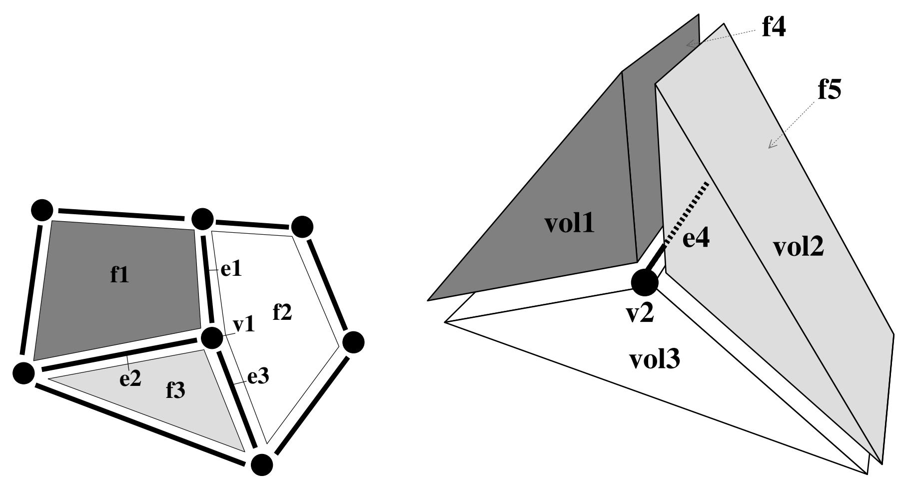


Figure 27.1: Example of subdivided objects that can be described by combinatorial maps. **Left**: A 2D object composed of three facets (2-cells), named *f1*, *f2* and *f3*, nine edges (1-cells) and seven vertices (0-cells). *f1* and *f2* are adjacent along edge *e1*, thus *e1* is incident both to *f1* and *f2*. Vertex *v1* is incident to edge *e1*, thus *v1* is incident to *f1* and *f2* by transitivity. **Right**: A 3D object (only partially represented for vertices and edges) composed of three volumes (3-cells), named *vol1*, *vol2* and *vol3*, twelve facets (2-cells) (there is one facet *f4* between *vol1* and *vol2*, and similarly between *vol1* and *vol3* and *vol2* and *vol3*), sixteen edges (1-cells), and eight vertices (0-cells). *vol1* and *vol2* are adjacent along facet *f4*, thus *f4* is incident both to *vol1* and *vol2*. Edge *e4* is incident to the three facets between *vol1* and *vol2*, *vol1* and *vol3*, and *vol2* and *vol3*. *e4* is also incident to the three volumes by transitivity.

A combinatorial map is an edge-centered data structure describing the cells and the incidence and adjacency relations, using only one basic element called *dart*, and a set of *pointers* between these darts. A dart can be thought as a part of an oriented edge (1-cell), together with a part of incident cells of dimensions 0, 2, 3,. . . , *d*. When a dart *d0* describes a part of an *i*-cell *c*, we say that *d0 belongs* to *c*, and that *c contains d0*. Let us look at the example in Figure 27.2 showing the 2D and 3D combinatorial maps describing the two objects given in Figure 27.1.

First let us start in 2D (Figure 27.2 (Left)). Facet *f1* is described by four darts. These darts are linked together with pointers. Starting from a dart and following a β_1 pointer, we get to a dart which belongs to the same facet

[1] A 2D combinatorial map is equivalent to a halfedge data structure: there is a one-to-one mapping between elements of both data structures, halfedges corresponding to darts.

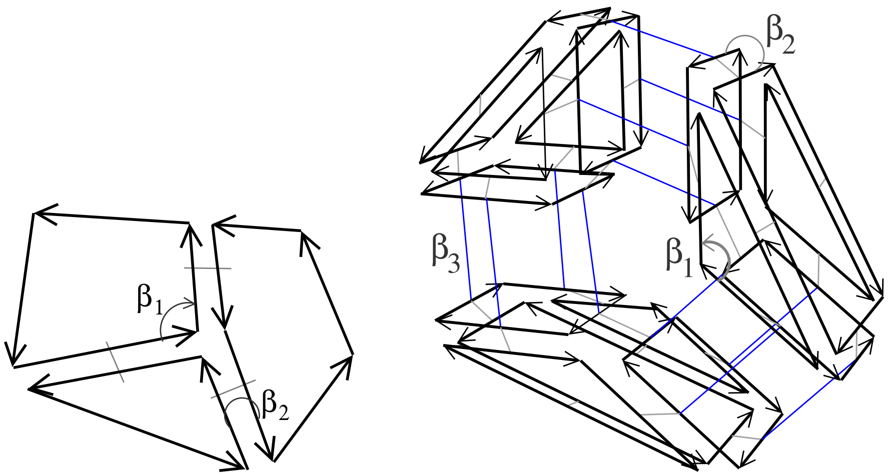


Figure 27.2: Combinatorial maps representing the objects given in Figure 27.1. **Left**: The 2D combinatorial map which contains 12 darts. **Right**: The 3D combinatorial map which contains 54 darts (18 for each volume).

but to the next edge (1-cell, which explains the index 1 of β_1). Starting from any dart and following β_1 pointers, we can reach exactly all the darts describing the facet. Starting from a dart and following a β_2 pointer, we get to a dart which belongs to the same edge but to the neighboring facet (2-cell, which explains the index 2 of β_2). Starting from any dart and following β_2 pointers, we can reach exactly all the darts describing the edge (in 2D one or two darts).

Things are slightly different for vertices. Indeed, each β_i points to a dart belonging to a different i-cell, but also to a different 0-cell (vertex). This is so because two linked darts have opposite orientations. For this reason, starting from any dart belonging to a vertex v, we have to follow β_2 then β_1 to reach exactly the darts describing the vertex v. In fact, by composing two β_is, we always obtain a dart belonging to the same vertex (if we do not start by following a β_1 pointer).

The main interest of combinatorial map definition based on darts and β_i pointers is to be able to increase the dimension "only" by adding new pointers. We can verify this fact by studying the 3D example (Figure 27.2 (Right)). In addition to β_1 and β_2 of the 2D case, there is a new pointer β_3.

If we take a closer look at the central edge *e4* shown in Figure 27.3 (Left), we can see that it is described by six darts linked together. Starting from a dart and following a β_3 pointer, we get to a dart which belongs to the same edge, to the same facet, but to the neighboring volume (a 3-cell, which explains the index 3 in β_3). Similarly, starting from a dart and following a β_2 pointer, we get to a dart which belongs to the same edge, to the same volume, but to the neighboring facet (2-cell). Starting from any of these six darts and following β_2 and β_3 pointers, we can reach exactly the six darts describing edge *e4*.

For facets, by following a β_1 pointer, we get to a dart which belongs to the same facet, to the same volume, but to the next edge (1-cell, which explains the index 1 of β_1). Starting from any dart and following β_1 and β_3 pointers, we can reach exactly all the darts describing the facet (see Figure 27.3 (Right)). For volumes, starting from any dart and following β_1 and β_2 pointers, we can reach exactly all the darts describing the volume.

For vertices, we have to follow β_2 then β_1, and β_3 then β_1 to reach exactly the darts describing the vertex v. Indeed, as in 2D, we have to compose two β_is to obtain a dart belonging to the same vertex (if we do not start by following a β_1 pointer).

In some cases, the general rule that by following a β_i we get a dart which belongs to the neighboring i-cell is not true, as for example for darts belonging to the boundary of the represented object. For example, in Figure 27.1 (Left), any dart *d0* that does not belong to edge *e1*, *e2* and *e3* belongs to a 2-cell, and there is no neighboring

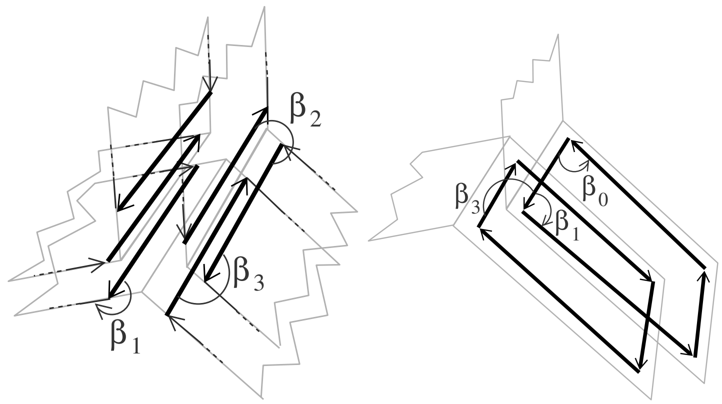


Figure 27.3: Two zooms on the 3D combinatorial map given in Figure 27.2 (Right). **Left**: Zoom around the central edge *e4* which details the six darts belonging to the edge. **Right**: Zoom around the facet between volumes *vol2* and *vol3* which details the eight darts belonging to the facet.

facet along the edge containing *d0*. Another example is in Figure 27.1 (Right), for any dart *d0* that belongs to facet *f5*. *d0* belongs to volume *vol2*, but there is no neighboring volume along this facet. The general rule is also not true for unbounded cells. For example if we remove a dart in Figure 27.2 (Left), we obtain an unbounded facet having a dart without next dart for β_1, and if we remove a facet in Figure 27.2 (Right), we obtain an unbounded volume having some darts without neighboring facet for β_2. In such a case, there is a particular value called $\varnothing$ used to describe that a dart *d0* is not linked to another dart in dimension *i*.

Combinatorial maps are defined in any dimension. A 0D combinatorial map is a set of isolated darts describing isolated vertices. A 1D combinatorial map describes paths or cycles of darts corresponding to paths or cycles of edges, and equivalent to double linked lists. The most useful cases are 2D and 3D combinatorial maps. Since 2D combinatorial maps are equivalent to halfedge data structure, notions are illustrated in 3D in the following examples to help the reader understand this specific case. But it is important to keep in mind that one main interest of combinatorial maps is their generic definition in any dimension, and that everything presented in this manual is valid in any dimension.

A *d*D combinatorial map is useful when you want to describe *d*D objects and the adjacency relations between these objects, and you want to be able to efficiency traverse these objects by using the different relations. For example, we can use a 3D combinatorial map to describe a 3D segmented image: each 3-cell corresponds to a region in the image and each 2-cell corresponds to a contact area between two regions.

A combinatorial map does not contain any geometrical information. However, this package allows to associate any information to the cells of the combinatorial map. A specific information, which is often used in practice, consists in adding linear geometry to a combinatorial map by associating a point to each vertex of the map[2]: this is the object of the *Linear_cell_complex* package. This package can for example be useful to describe 3D buildings as set of walls, rooms, doors and windows (both combinatorial and geometrical descriptions) and all the adjacency relations between these elements allowing for example to move a camera in a given building from rooms to rooms by traversing doors.

[2]When an object has a point associated to each vertex, each edge is thus a straight line segment, which explains the name ”linear geometry”.

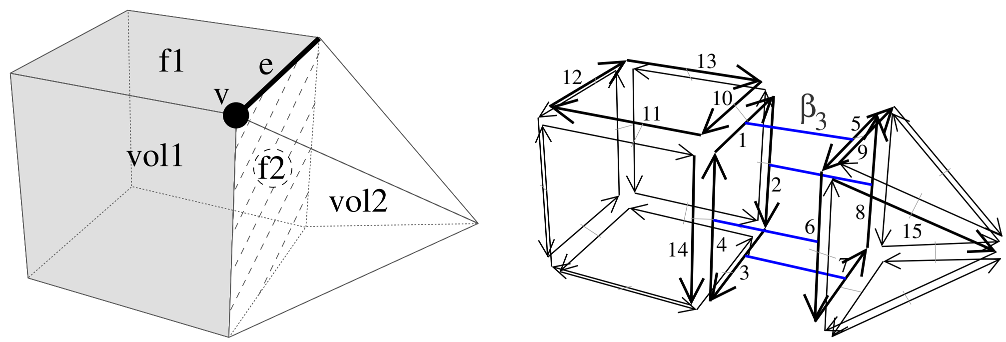


Figure 27.4: Example of a 3D combinatorial map. **Left**: A 3D object made of two volumes adjacent along facet *f2*. **Right**: The corresponding 3D combinatorial map. Darts are drawn with arrows, sometimes numbered. Two darts linked by β_1 are drawn consecutively (for example $\beta_1(10)$=11), and two darts linked by β_2 are drawn parallel, in reverse orientations, with a little gray segment joining them (for example $\beta_2(1)$=10). β_3 pointers are represented by blue segments (for example $\beta_3(1)$=5).

27.2 Data Structure Presentation

In this section, we describe *d*D combinatorial maps in terms of data structure and operations. Mathematical definitions are provided in Section 27.7, and a package description is given in Section 27.3.

27.2.1 Combinatorial Map and Darts

A *d*D combinatorial map is a set of darts *D*. A dart *d0* is an element that can be *linked* with *d*+1 darts by pointers called β_i, with $0 \leq i \leq d$. Dart *d0* is said *i-free* when $\beta_i(d0)=\varnothing$. Each β_i, for $2 \leq i \leq d$, is its own inverse, i.e., if dart *d0* is not *i*-free, then $\beta_i(\beta_i(d0))=d0$. This is different for β_0 and β_1: β_0 is the inverse of β_1, i.e., if darts *d1* and *d2* are such that $\beta_1(d1)=d2$, then $\beta_0(d2)=d1$. Given dart *d1*, if there is no dart *d2* such that $\beta_1(d2)=d1$, then $\beta_0(d1)=\varnothing$. $\varnothing$ is a constant, which does not belong to the set of darts *D* of the combinatorial map. However, by definition $\varnothing$ is linked with itself for all β_is: $\forall i$, $0 \leq i \leq d$, $\beta_i(\varnothing)=\varnothing$.

A combinatorial map is *without i-boundary* if there is no *i*-free dart, and it is *without boundary* if it is without *i*-boundary for all dimensions $1 \leq i \leq d$.

We show in Figure 27.4 a 3D object and the corresponding 3D combinatorial map. This map has 40 darts represented by arrows, some darts being numbered. In this combinatorial map, we have for example $\beta_1(1)$=2, $\beta_2(1)$=10, and $\beta_3(1)$=5. This combinatorial map is without 1-boundary and 2-boundary, but has some 3-boundary, because some darts are 3-free, for example $\beta_3(10)=\varnothing$ and $\beta_3(12)=\varnothing$.

27.2.2 Cells as Sets of Darts

A cell in a *d*D combinatorial map is implicitly represented by a subset of darts. In this section, we will see how to retrieve all cells containing a given dart, how to retrieve all darts belonging to a cell containing a given dart, and how incidence and adjacency relations are defined in terms of darts.

The first important property of a combinatorial map is that each dart belongs to an *i*-cell, $\forall i$, $0 \leq i \leq d$. For example in 3D, a dart belongs to a vertex, an edge, a facet, and a volume. This means that a 3D combinatorial map containing an isolated dart contains exactly one vertex, one edge, one facet and one volume.

The second important property is that cells of a combinatorial map correspond to specific *orbits*. Given a set $S \subseteq \{\beta_1,\ldots,\beta_d\}$ and a dart *d0*, the *orbit* $\langle S\rangle(d0)$ is the set of darts that can be reached from *d0* by following any combination of any β_i's in S and their inverses (to simplify notations, we can use for example $\langle\beta_1,\beta_4\rangle(d0)$ to denote $\langle S\rangle(d0)$ with $S=\{\beta_1,\beta_4\}$).

Given a dart *d0*, in general, $\beta_i(d0)$ (with $1\leq i\leq$d) belongs to the same cells as *d0*, only the *i*-cell and 0-cell are different. There are two exceptions: (1) if *d0* is *i*-free, then $\beta_i(d0)=\varnothing$; (2) if $\beta_i(d0)$ belongs to the same *i*-cell as *d0* (case of multi-incidence). For example if an edge is an isolated loop, it is incident twice to the same vertex, then given a dart *d0* belonging to this edge, $\beta_1(d0)$ goes to the next edge, which is in fact the same edge.

Since $\beta_i(d0)$ (with $1\leq i\leq d$) allows to change the current *i*-cell, all the darts that can be reached from *d0* by using any combination of β_j's, $\forall j$, $1\leq j\leq d$ and $j\neq i$ and their inverse are contained in the same *i*-cell as *d0*. The *i*-cell containing *d0* is defined in terms of orbit by $\langle\beta_1,\ldots,\beta_{i-1},\beta_{i+1},\ldots,\beta_d\rangle(d0)$.

There is a special case for vertices. Given a dart *d0*, the set of darts contained in the same vertex as *d0* are the darts that can be reached from *d0* by using any combination of $\beta_i\circ\beta_j$, $\forall i,j$, $1\leq i<j\leq d$, and their inverse. The 0-cell containing *d0* is defined in terms of orbit by $\langle\{\beta_i\circ\beta_j|\forall i,j\text{: } 1\leq i<j\leq d\}\rangle(d0)$.

Orbit $\langle\beta_1,\ldots,\beta_d\rangle(d0)$ is the *connected component* containing dart *d0*. A combinatorial map is *connected* if this set is equal to the set of all the darts of the combinatorial map.

A last important property of cells is that for all dimensions *i* the set of *i*-cells forms a partition of the set of darts *D*, i.e. for any *i*, the union of the sets of darts of all the *i*-cells is equal to *D*, and the sets of darts of two different *i*-cells are disjoint.

Let us give some examples of cells in 3D, for the 3D combinatorial map of Figure 27.4:

- All the darts belonging to the same edge can be obtained by any combination of β_2 and β_3: for example edge *e* of the object corresponds in the combinatorial map to the set of darts {1,5,9,10}. Given any dart belonging to this edge, we retrieve all the other darts by, for example, a breadth-first traversal. In terms of orbits, this 1-cell corresponds to $\langle\beta_2,\beta_3\rangle(1)$.
- All the darts belonging to the same facet can be obtained by any combination of β_1 and β_3: for example facet *f2* corresponds in the combinatorial map to the set of darts {1,2,3,4,5,6,7,8}. Facet *f1* corresponds to the set of darts {10,11,12,13}. Note that these last darts are 3-free since there is no other volume sharing this facet. In terms of orbits, *f2* corresponds to $\langle\beta_1,\beta_3\rangle(1)$ and *f1* corresponds to $\langle\beta_1,\beta_3\rangle(10)$.
- All the darts belonging to the same volume can be obtained by any combination of β_1 and β_2: for example volume *vol1* corresponds in the combinatorial map to the set of the twenty-four darts representing the cube. In terms of orbits, *vol1* corresponds to $\langle\beta_1,\beta_2\rangle(1)$.
- All the darts belonging to the same vertex can be obtained by any combination of $\beta_1\circ\beta_2$, $\beta_1\circ\beta_3$ and $\beta_2\circ\beta_3$ and their inverse functions. In our example, vertex *v* of the object corresponds in the combinatorial map to the set of darts {1,6,9,11,14,15}. Starting from dart 1, we obtain for example dart $14=(\beta_1\circ\beta_2)^{-1}(1)=\beta_2\circ\beta_0(1)$, dart $11=\beta_1\circ\beta_2(1)$, and dart $9=\beta_2\circ\beta_3(1)$. Intuitively, the set of darts corresponding to a vertex contains all the darts represented by arrows starting from this vertex. In terms of orbits, *v* corresponds to $\langle\beta_1\circ\beta_2,\ \beta_1\circ\beta_3,\ \beta_2\circ\beta_3\rangle(1)$.

Using this definition of cells as sets of darts, we can retrieve all the incidence and adjacency relations between the cells of the subdivision in a combinatorial map. Two cells are *incident* if the intersection of their two sets of darts is non empty (whatever the dimension of the two cells). Two *i*-cells *c1* and *c2*, $1\leq i\leq d$, are *adjacent* if there is $d1\in c1$ and $d2\in c2$ such that $d1=\beta_i(d2)$ (or $d2=\beta_i(d1)$ for i=1).

In the example of Figure 27.4, vertex *v* and edge *e* are incident since the intersection of the two corresponding sets of darts is $\{1,9\}\neq\emptyset$. Vertex *v* is incident to facet *f2* since the intersection of the two corresponding sets of darts is $\{1,6\}\neq\emptyset$. Edge e and facet $f1$ are incident since the intersection of the two corresponding sets of darts is $\{10\}\neq\emptyset$. Finally, facets $f1$ and $f2$ are adjacent because $10\in f1$, $1\in f2$ and $10=\beta_2(1)$.

We can consider i-cells in a dimension d' with $i\leq d'\leq d$. The idea is to consider the i-cells as if the combinatorial map was in d' dimension. For that, we only take into account the β_js for $j\leq d'$. The i-cell containing $d0$ in dimension d' is the orbit $\langle\beta_1,\ldots,\beta_{i-1},\beta_{i+1},\ldots,\beta_{d'}\rangle(d0)$, and the 0-cell is the orbit $\langle\{\beta_i\circ\beta_j|\forall i,j\colon 2\leq i<j\leq d'\}\rangle(d0)$. By default, i-cells are considered in dimension d, the dimension of the combinatorial map.

In the example of Figure 27.4, the 2-cell containing dart 1 is facet $f2$ which is the set of darts $\{1,2,3,4,5,6,7,8\}$. If we consider the same 2-cell in dimension 2, we obtain the set of darts $\{1,2,3,4\}$. Intuitively we "forget" β_3 and we obtain the set of darts of the facet containing dart 1 restricted to the volume containing this dart.

27.2.3 How to Associate Information to Cells

Combinatorial maps only describe the cells of the subdvision, and all the incidence and adjacency relations between these cells. This is not enough for many applications which need to associate *information* to cells. This can be geometric or non-geometric information, such as 3D points associated to vertices, the edge length associated to edges, or a color or normal to a facet.

To answer this need, a combinatorial map allows to create *attributes* which are able to store any information, and to associate attributes to cells of the combinatorial map. We denote i-attributes for the attributes associated with i-cells. Attributes may exist for only some of the dimensions, and if they exist for dimension i, they do not necessarily exist for each of the i-cells. More precisely, i-attributes are associated to i-cells by an injection:

- two different i-cells are associated to two different i-attributes;
- an i-cell may have no associated i-attribute.

Since i-cells are not explicitely represented in combinatorial maps, the association between i-cells and i-attributes is transferred to darts: if attribute a is associated to i-cell c, all the darts belonging to c are associated to a.

We can see two examples of combinatorial maps having some attributes in Figure 27.5. In the first example (Left), a 2D combinatorial map has 1-attributes containing a float, for example corresponding to the length of the associated 1-cell, and 2-attributes containing a color in RGB format. In the second example (Right), a 3D combinatorial map has 2-attributes containing a color in RGB format.

27.2.4 Combinatorial Map Properties

There are some conditions that a combinatorial map must satisfy to be valid. Some of them have already been given about the β pointers (see Section 27.2.1) and about the association between darts and attributes (see Section 27.2.3).

There is an additional condition related to the type of represented objects, which are *quasi-manifold* orientable dD objects. A dD quasi-manifold is an object obtained by taking some isolated d-cells, and allowing to glue d-cells along *(d-1)*-cells. It is orientable if it is possible to embed it in the Euclidean space and to define a global "left" and "right" direction in each point of the embedded object. In 2D, quasi-manifolds are manifolds, but this is no longer true in higher dimension as we can see in the example presented in Figure 27.6. In this example, the object to the right is not a manifold since the neighborhood of the point p in the object is not homeomorphic[3] to a 3D ball.

[3]Intuitively, two objects are homeomorphic if each object can be continuously deformed into the second one. In such a case, the two objects have exactly the same topological properties.

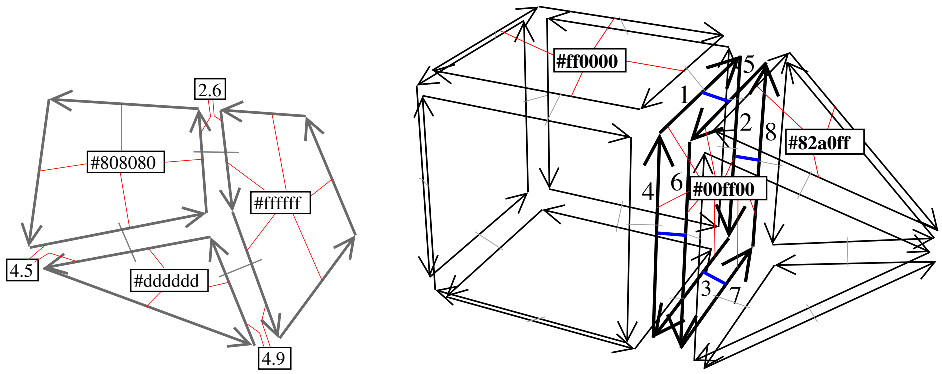


Figure 27.5: Example of combinatorial maps with attributes. Attributes are represented by black rectangles containing an information, and association between darts and attributes are represented by small lines. **Left**: A 2D combinatorial map with 1-attributes containing a double, for example corresponding to the length of the 1-cell, and 2-attributes containing a color in RGB format. Only three edges of the combinatorial map, among the nine, are associated to a 1-attribute. All the 2-cells are associated to a 2-attribute. **Right**: A 3D combinatorial map with 2-attributes containing a color in RGB format. Only three 2-cells of the combinatorial map, among the ten, are associated to a 2-attribute.

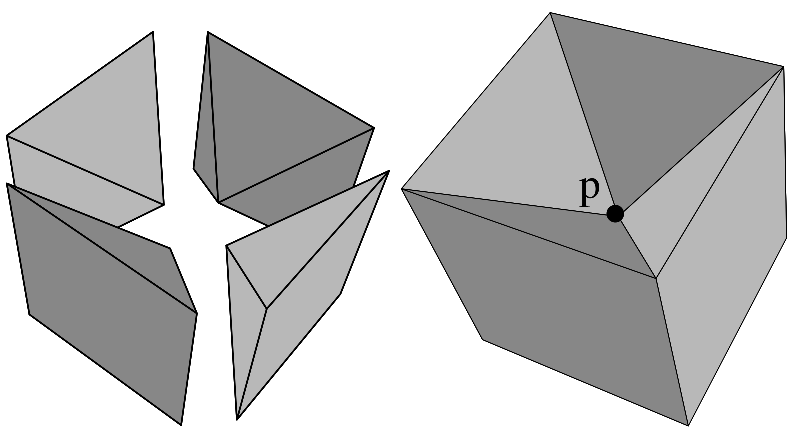


Figure 27.6: Example of a 3D quasi-manifold which is not a manifold. The object to the right is made of the four pyramids (shown to the left) glued together along facets, thus it is a quasi-manifold.

Combinatorial maps can only represent quasi-manifolds due to the definition of β pointers. As we have seen in Section 27.2.2, $\beta_i(d0)$ (with $1 \le i \le d$) belongs to the same cells as *d0*, only the *i*-cell and 0-cell are different. In other words, β_i links two *i*-cells that share a common *(i-1)*-cell: it is not possible to link more than two *i*-cells along a same *(i-1)*-cell. For this reason, it is not possible to describe non quasi-manifold objects as those shown in Figure 27.7 by combinatorial maps.

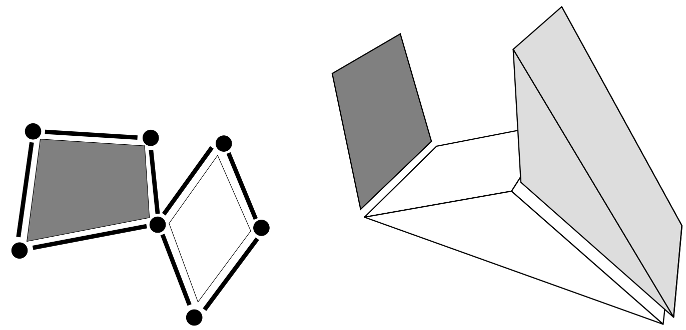


Figure 27.7: Three examples of non quasi-manifold objects. **Left**: A 2D object which is not a quasi-manifold since the two 2-cells share a common vertex but no common 1-cell. **Middle**: A 3D object which is not a quasi-manifold since is it not only composed by 3D cells glued together (there is an isolated 2-cell in dark gray). **Right**: A 3D object which is not a quasi-manifold since the two 3-cells share a common edge but no common 2-cell.

Due to this additional condition, any objects can not be represented by a combinatorial map but only orientable quasi-manifolds. We need to study now the inverse relation. Does any set of darts linked together by β_i's, with $0 \le i \le d$ correspond to a quasi-manifold? As we can see in Figure 27.8, the answer is no.

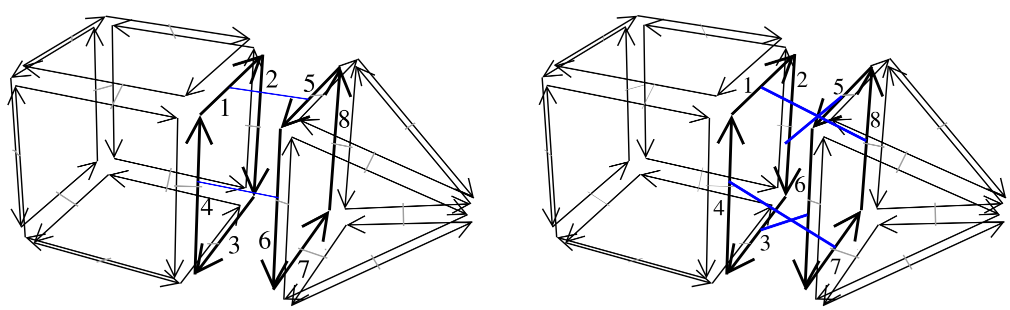


Figure 27.8: Two examples of darts linked together by some β_0, β_1, β_2 and β_3 which does not represent a 3D quasi-manifold, and thus which are not 3D combinatorial map. **Left**: In this example, all the darts are 3-free except $\beta_3(1)=5$ and $\beta_3(4)=6$ (and vice-versa). **Right**: In this example, darts linked by β_3 are not in the same order in both 3-cells.

In the first example (Left), there are two 3-cells (one to the left for the cube, a second to the right for the pyramid) which are "partially adjacent" along one 2-cell. Indeed, only two darts of the 2-cell are linked by β_3. We have $\beta_3(1)=5$ and $\beta_3(4)=6$ (and reciprocally). This configuration is not possible in a quasi-manifold: two *d*-cells are always glue along an "entire" *(d-1)*-cells.

But as we can see in the second example (Right), the condition that all the darts of the cell are linked in not sufficient. Indeed, in this example, all the darts of the 2-cell between the cube and the pyramid are linked together by β_3. However, this configuration does not correspond to an orientable 3D quasi-manifold. Indeed, the operation of gluing two *d*-cells along one *(d-1)*-cell must preserve the initial *(d-1)*-cell.

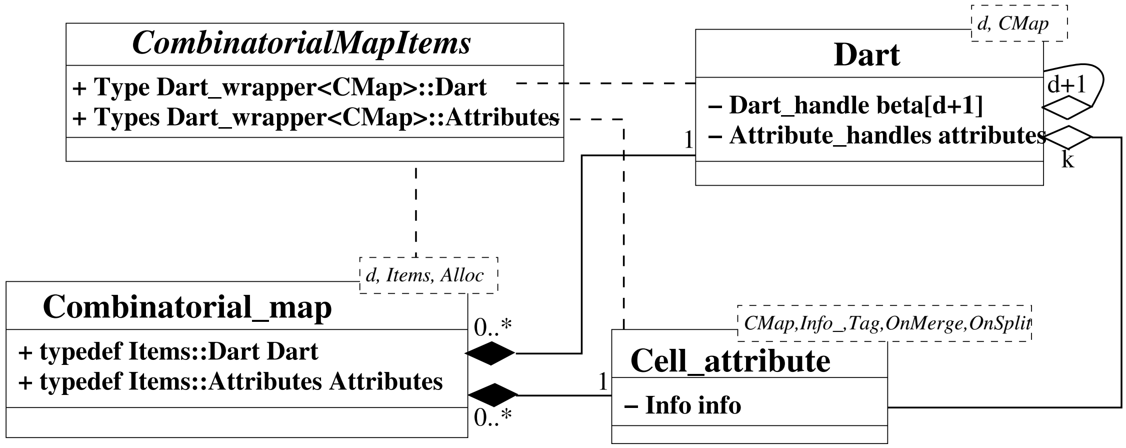


Figure 27.9: UML diagram of the main classes of the package. k is the number of non void attributes.

To avoid these two kinds of configurations, conditions are added on β pointers compositions (see Section 27.7, condition (4) of the definition of combinatorial maps). Intuitively these conditions say that if two darts are linked by β_i, then all the required darts are linked by β_i two by two in such a way that neighborhood relations are preserved.

We say that a combinatorial map is *valid* if it satisfies all the conditions on β pointers and on association between darts and attributes. High level operations provided on combinatorial maps ensure that these conditions are always satisfied. Sometimes, it can be useful to use low level operations in a specific algorithm, for example to modify locally a combinatorial map in a really fast way. In such a case, additional operations may be needed to restore these validity conditions.

27.3 Software Design

The diagram in Figure 27.9 shows the different classes of the package. *Combinatorial_map* is the main class (see Section 27.3.1). It allows to manage darts (see Section 27.3.3) and attributes (see Section 27.3.4). Users can customize a combinatorial map thanks to an items class (see Section 27.3.2), which defines the dart type and the attribute types. These types may be different for different dimensions, and they may also be void. The darts and attributes are accessed through *handles*. A handle is a model of the *Handle* concept, thus supporting the two dereference operators *operator** and *operator->*.

27.3.1 Combinatorial Maps

The class *Combinatorial_map<d,Items,Alloc>* is a model of the *CombinatorialMap* concept. It has three template parameters standing for the dimension of the combinatorial map (an *unsigned int*), an items class (a model of the *CombinatorialMapItems* concept), and an allocator which must be a model of the allocator concept of the STL. Default classes are provided for the items and the allocator classes.

The main role of the class *Combinatorial_map* is the storage and the management of darts. It allows to create or remove an isolated dart from the combinatorial map. The *Dart_handle* type defines a handle to the type of used darts (given in the items class). *Combinatorial_map* provides several *ranges* which allow to iterate over specific subsets of darts of the combinatorial map (see Section 27.4.1). It also defines several methods to link and to unlink darts by β_is (see Section 27.5.1). We said that a dart *d0* is *i*-free if $\beta_i(d0)=\varnothing$. The ∅

constant is represented in the class *Combinatorial_map* through a *static const Dart_handle* called *null_dart_handle*. Finally, some high level operations are defined as global functions taking a *Combinatorial_map* as argument (see Section 27.5.2)

The second role of the class *Combinatorial_map* is the storage and the management of attributes. It allows to create or remove an attribute, and provides methods to associate attributes and cells. A range is defined for each *i*-attribute allowing to iterate over all the *i*-attributes of the combinatorial map. Finally, *Combinatorial_map* defines several types allowing to manage the attributes. We can use *Combinatorial_map::Attribute_handle<i>::type* for a handle to the *i*-attributes (and the const version *Combinatorial_map::Attribute_const_handle<i>::type*) and *Combinatorial_map::Attribute_type<i>::type* for the type of the *i*-attributes.

27.3.2 Combinatorial Map Items

The *CombinatorialMapItems* concept defines dart and attribute types of a combinatorial map. It contains one inner class named *Dart_wrapper*, having one template parameter, *CMap*, a model of *CombinatorialMap* concept. The *Dart_wrapper<CMap>* class provides two local types: *Dart* which must be a model of the *Dart* concept, and *Attributes* which defines the attributes and their types.

The *Attributes* tuple must contain at most d+1 types (one for each possible cell dimension of the combinatorial map). Each type of the tuple must be either a model of the *CellAttribute* concept or *void*. The first type corresponds to 0-attributes, the second to 1-attributes and so on. If the i^{th} type in the tuple is *void*, *(i-1)*-attributes are disabled: we say that *(i-1)*-attributes are *void*. Otherwise, *(i-1)*-attributes are enabled and have the given type: we say *(i-1)*-attributes are *non void*. If the size of the tuple is k, with $k<dimension+1$, $\forall i$: $k \leq i \leq$dimension, i-attributes are void.

The class *Combinatorial_map_min_items<d>* is a model of the *CombinatorialMapItems* concept which can be used for default behaviors. It defines *CGAL::Dart<d,CMap>* as type of dart, and *Attributes* as empty tuple.

27.3.3 Darts

The class *Dart<d,CMap>*, a model of the *Dart* concept, defines a dD dart. It has two template parameters standing for the dimension of the combinatorial map, and a model of the *CombinatorialMap* concept, which provides the two types *Dart_handle* and *Dart_const_handle*.

Each instance *d0* of *Dart<d,CMap>* stores the β_i pointers in an array of d+1 *Dart_handle* (because we describe also the β_0 pointer). It also stores the attributes associated to this dart in a tuple of *CMap::Attribute_handle<i>::type*, one for each non void *i*-attribute.

Methods are defined allowing to retrieve each β_i and each associated *i*-attribute of *d0*, and allowing to test if *d0* dart is *i*-free.

Note that the use of the *Dart* class is not hard wired in the combinatorial map class. Users can provide their own model of the *Dart* concept, and pass it to the combinatorial map with the help of a custom item class.

27.3.4 Cell Attributes

The class *Cell_attribute<CMap,Info_,Tag,OnMerge,OnSplit>*, a model of the *CellAttribute* concept, represents an attribute associated with a cell of a combinatorial map. The template parameter *CMap* must be a model of the *CombinatorialMap* concept. The attribute stores a handle to one dart of its associated cell when the template parameter *Tag* is *Tag_true*. *Info_* is the type of information stored in the attribute. It may be *void*. *OnMerge* and *OnSplit* must be either *Null_functor*, or models of the *Binary Function* concept having two references to a model of *CellAttribute* as type of both parameters and *void* as return type. There are two default parameters for *OnMerge* and *OnSplit*, which are *Null_functor*, a default parameter for *Tag* which is *Tag_true*, and a default parameter for *Info_* which is *void*.

If *Info_* is different from *void*, the class *Cell_attribute* contains two methods *info()* returning the information contained in the attribute (const and non const version). The information is returned by reference, thus the non const version allows the modification of the information.

Two attributes are merged when their corresponding cells are merged into one cell during some operation. In this case, the functor *OnMerge* is called, unless it is equal to *Null_functor*. This functor allows the user to define its own custom behavior when two attributes are merged (for example if the information is a color, we can compute the average color of the two initial attributes, and affect this value to the first attribute, see example in Section 27.6.4). Similarly, the functor *OnSplit* is called when one attribute is split in two, because its corresponding cell is split in two during some operation, unless it is equal to *Null_functor*. In any high level operation, *OnMerge* is called before to start the operation (i.e. before modifying the combinatorial map), and *OnSplit* is called when the operation is finished (i.e. after all the modifications were made).

What we said for the dart also holds for the cell attribute. The combinatorial map can be used with any user defined model of the *CellAttribute* concept.

27.3.5 Example of Combinatorial Map Definition

Here comes an example of two combinatorial map definitions. The first case *Example_cmap4* defines a 4D combinatorial map which uses all the default values (*Dart* and *Combinatorial_map_min_items*). The second example *Example_custom_cmap3* uses its own model of the *CombinatorialMapItems* concept. In this model, the type of dart is *Dart<3,CMap>*, thus a dart is in 3D, and an attribute containing an integer is associated to edges.

```
typedef CGAL::Combinatorial_map<4> Example_cmap4;

struct Example_items_3
{
   template <class CMap>
   struct Dart_wrapper
   {
     typedef CGAL::Dart<3, CMap> Dart;
     typedef CGAL::Cell_attribute<CMap, int> Edge_attrib;
     typedef CGAL::cpp0x::tuple<void,Edge_attrib> Attributes;
   };
};
typedef CGAL::Combinatorial_map<3, Example_items_3> Example_custom_cmap3;
```

27.4 Iteration and Creation Operations

An important operation in combinatorial maps consists in iterating over specific subsets of darts or over attributes. For that, several *ranges* are offered (see Section 27.4.1). A range is a model of the *Range* concept, thus supporting the two methods *begin()* and *end()* allowing to iterate over all the elements in the range. Several global functions allow to create specific configurations of darts into a combinatorial map (see Section 27.4.2). Darts can be marked during operations, for example when performing a breadth-first search traversal, thanks to

Boolean marks (see Sections 27.4.3). In the following, we denote by *dh0*, *dh1*, *dh2* the dart handles for the darts *d0*, *d1*, *d2*, respectively. That is *d0 == *dh0*.

27.4.1 Iterating over Orbits, Cells, and Attributes

The combinatorial map offers iterators to traverse the darts of a specific orbit, to traverse all darts of one cell, or one dart per cell, and to traverse all *i*-attributes.

Instead of the *begin()/end()* member function pair as we know it from STL containers, and from most CGAL data structures, the combinatorial map defines range classes which are all models of the *Range* concept.

There are three different categories of dart range classes:

- *Dart_range*: range of all the darts of a combinatorial map;
- *Dart_of_orbit_range<Beta...>*: range of all the darts of the orbit $\langle$*Beta...*$\rangle$(*d0*) for a given *d0*. *Beta...* is a sequence of integers $i_1,\ldots,i_k$, each $i_j \in \{0,\ldots,d\}$. These integers must satisfy: $i_1 < i_2 < \ldots < i_k$, and ($i_1 \neq 0$ or $i_2 \neq 1$) (for example *Dart_of_orbit_range<1,2>* for the orbit $\langle\beta_1,\beta_2\rangle$(*d0*));
- *Dart_of_cell_range<i,dim>*: range of all the darts of the *i*-cell containing a given dart. The *i*-cell is considered in dimension *dim* (with $0 \leq dim \leq d$, *dim*=*d* by default), with $0 \leq i \leq dim+1$. If *i=dim+1*, *Dart_of_cell_range<i,dim>* is the range of all the darts of the connected component containing a given dart.

There are also two different classes of ranges containing one dart per *i*-cell. Note that in these classes, the dart of each *i*-cell can be any dart of the cell. Moreover, each *i*-cell (and *j*-cell in the second case) is considered in dimension *dim* (with $0 \leq dim \leq d$, *dim*=*d* by default).

- *One_dart_per_cell_range<i,dim>*: range containing one dart of each *i*-cell of the combinatorial map, $0 \leq i \leq dim+1$ (for example *One_dart_per_cell_range<2>* for the range of one dart per 2-cell of the combinatorial map);
- *One_dart_per_incident_cell_range<i,j,dim>*: range containing one dart of each *i*-cell incident to the *j*-cell containing a given dart, with $0 \leq i \leq dim+1$ and $0 \leq j \leq dim+1$ (for example *One_dart_per_incident_cell_range<0,3>* for the range of one dart per vertex of the volume incident to the starting dart). If *i=j*, the range contains only the given dart.

The iterators of the *Dart_range* are bidirectional iterators, while the iterators of the other four ranges are forward iterators. The value type of all these iterators is *Dart* thus all these iterators can be directly used as *Dart_handle*.

Additionally, there is a range over non void *i*-attributes: *Attribute_range<i>::type*, having a bidirectional iterator with value type *Attribute_type<i>::type*.

For each range, there is an associated const range, a model of the *ConstRange* concept. You can find some examples of ranges in Section 27.6.1.

27.4.2 Construction Operations

Several global functions allow to create specific configurations of darts into a combinatorial map. Existing darts in the combinatorial map are not modified. Note that the dimension of the combinatorial map must be large enough: darts must contain all the β pointers used by the operation. All these functions take an instance of *CombinatorialMap* as first parameter (called *cm*) and return a *Dart_handle* to a new dart created during the operation.

- *make_edge<CMap>(cm)*: creates an isolated edge (two darts linked by β_2); dimension must be greater or equal than two;
- *make_combinatorial_polygon<CMap>(cm,lg)*: creates an isolated combinatorial polygon of length *lg* (*lg* darts linked by β_1), for *lg>0*; dimension must be greater or equal than one;
- *make_combinatorial_tetrahedron<CMap>(cm)*: creates an isolated combinatorial tetrahedron (four combinatorial triangles linked together by β_2); dimension must be greater or equal than two;
- *make_combinatorial_hexahedron<CMap>(cm)*: creates an isolated combinatorial hexahedron (six combinatorial quadrangles linked together by β_2); dimension must be greater or equal than two.

advanced

27.4.3 Boolean Marks

It is often necessary to mark darts, for example to retrieve in *O(1)* if a given dart was already processed during a specific algorithm, for example, iteration over a given range. Users can also mark specific parts of a combinatorial map (for example mark all the darts belonging to objects having specific semantics). To answer these needs, a *CombinatorialMap* has a certain number of Boolean marks (fixed by the constant *NB_MARKS*). When one wants to use a Boolean mark, the following methods are available (with *cm* an instance of a combinatorial map):

- get a new free mark: *int m = cm.get_new_mark()* (return -1 if no mark is available);
- set mark *m* for a given dart *d0*: *cm.mark(dh0,m)*;
- unset mark *m* for a given dart *d0*: *cm.unmark(dh0,m)*;
- test if a given dart *d0* is marked for *m*: *cm.is_marked(dh0,m)*;
- unmark all the darts of *cm* for *m*: *cm.unmark_all(m)*;
- negate mark *m* of all the darts of *cm*: *cm.negate_mark(m)*. All the marked darts become unmarked and all the unmarked darts become marked;
- free mark *m*: *cm.free_mark(m)*. This method unmarks all the darts of *cm* for *m* before freeing it.

It is important to free a mark when it is no longer needed, otherwise you may at some point run out of marks.

The following example illustrates how to use marks. Two combinatorial tetrahedra are created and 3-sewn (see Section 27.5.1 for a detailed description of the sew operation). Then a mark is reserved and used to mark all the darts belonging to the first combinatorial tetrahedron. Finally, these tetrahedron are merged. The marks allow us to know which darts come from the first and second tetrahedron.

```
#include <CGAL/Combinatorial_map.h>
#include <CGAL/Combinatorial_map_constructors.h>
#include <CGAL/Combinatorial_map_operations.h>
#include <iostream>
#include <cstdlib>

typedef CGAL::Combinatorial_map<3> CMap_3;
typedef CMap_3::Dart_handle Dart_handle;

int main()
```

```
{
  CMap_3 cm;

  // 1) Reserve a mark.
  int mark = cm.get_new_mark();
  if ( mark==-1 )
    {
      std::cerr<<"No more free mark, exit."<<std::endl;
      exit(-1);
    }

  // 2) Create two tetrahedra.
  Dart_handle dh1 = make_combinatorial_tetrahedron(cm);
  Dart_handle dh2 = make_combinatorial_tetrahedron(cm);

  // 3) 3-sew them.
  cm.sew<3>(dh1, dh2);

  // 4) Mark the darts belonging to the first tetrahedron.
  for  (CMap_3::Dart_of_cell_range<3>::iterator
          it(cm.darts_of_cell<3>(dh1).begin()),
          itend(cm.darts_of_cell<3>(dh1).end()); it!=itend; ++it)
    cm.mark(it, mark);

  // 4) Remove the common 2-cell between the two cubes:
  // the two tetrahedra are merged.
  CGAL::remove_cell<CMap_3, 2>(cm, dh1);

  // 5) Thanks to the mark, we know which darts come from the first
   tetrahedron.
  unsigned int res=0;
  for (CMap_3::Dart_range::iterator it(cm.darts().begin()),
         itend(cm.darts().end()); it!=itend; ++it)
    {
      if ( cm.is_marked(it, mark) )
        ++res;
    }

  std::cout<<"Number of darts from the first tetrahedron:
   "<<res<<std::endl;
  cm.free_mark(mark);

  return EXIT_SUCCESS;
}
```

File: examples/Combinatorial_map/map_3_marks.cpp

advanced

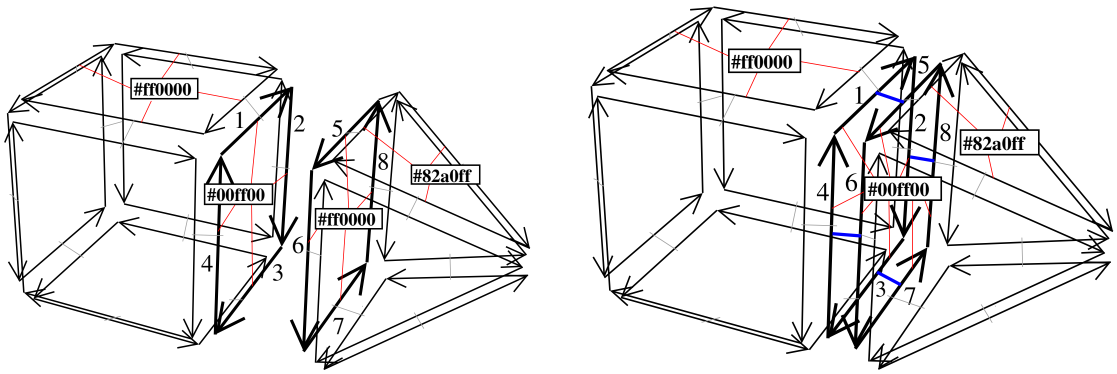


Figure 27.10: Example of 3-sew operation. **Left**: A 3D combinatorial map containing two volumes that are not connected, with 2-attributes. Each attribute contains a color in RGB format, and there are four 2-cells associated with attributes. Associations between darts and attributes are drawn with red segments. **Right**: The 3D combinatorial map obtained as result of *sew<3>(1,5)* (or *sew<3>(2,8)*, or *sew<3>(3,7)*, or *sew<3>(4,6)*). Darts (1,5), (2,8), (3,7) and (4,6) are linked together by β_3. The two 2-cells *c1*={1,2,3,4} and *c2*={5,6,7,8} are merged after the sew into the 2-cell {1,2,3,4,5,6,7,8}. We are in the case where the two attributes are non NULL, thus the first one is kept, and all the darts of *c2* are associated with the first attribute.

27.5 Modification Operations

Several operations allow to modify a given combinatorial map. There are two main categories of modification operations:

- Sew, link, unsew and unlink which modify some existing β pointers, without creating or removing darts (see Section 27.5.1);
- Removal and insertion of cells which modify both darts and β pointers (see Section 27.5.2).

27.5.1 Sewing Orbits and Linking Darts

The *CombinatorialMap* defines two groups of methods to modify the β pointers of existing darts.

- The *sew* and *unsew* methods iterate over two orbits in order to link or unlink specific darts two by two. Intuitively, a *sew<i>* operation glues two *i*-cells by identifying two of their *(i-1)*-cells (see example in Figure 27.10 where *sew<3>* is used to glue two 3-cells along one 2-cell). Reciprocally, a *unsew<i>* operation un-glues two *i*-cells which were glued along one of their *(i-1)*-cells. These methods guarantee that given a valid combinatorial map and a possible operation we obtain a valid combinatorial map as result of the operation.
- *advanced*

  The *link_beta* and *unlink_beta* methods only modify the pointer of two darts: the obtained combinatorial maps may be not valid. These operations can be useful to use low level operations in a specific algorithm, for example to modify locally a combinatorial map in a really fast way. In such a case, additional operations may be needed to restore the validity conditions.

  advanced

Linking two darts *d1* and *d2* by β_i, with $2 \leq i \leq d$ and $d1 \neq d2$, consists in modifying two β_i pointers such that $\beta_i(d1)=d2$ and $\beta_i(d2)=d1$. For i=1, the modification is $\beta_1(d1)=d2$ (and thus $\beta_0(d2)=d1$ by definition of β_0); in this case we can have $d1=d2$ (a dart linked with itself corresponds to an edge which is a loop).

Reciprocally, unlinking a given dart *d0* by β_i, with $2 \leq i \leq d$, consists in modifying two β_i pointers such that $\beta_i(\beta_i(d0))=\varnothing$ and $\beta_i(d0)=\varnothing$. For $i=1$, the modification is $\beta_1(d0)=\varnothing$ (and thus $\beta_0(\beta_1(d0))=\varnothing$ by definition of β_0). Note that is it possible to unlink a given dart for β_i only if it is not *i*-free.

The *sew<i>(dh1,dh2)* method consists mainly to link two by two several darts by β_i. This operation is possible only if there is a bijection f between all the darts of the orbit $D1=\langle\beta_1,\ldots,\beta_{i-2},\beta_{i+2},\ldots,\beta_d\rangle(d1)$ and $D2=\langle\beta_1,\ldots,\beta_{i-2},\beta_{i+2},\ldots,\beta_d\rangle(d2)$ satisfying: $f(d1)=d2$, and for all $e \in D1$, for all $j \in \{1,\ldots,i\text{-}2,i\text{+}2,\ldots,d\}$, $f(\beta_j(e))=\beta_j^{-1}(f(e))$. Intuitively, this condition ensures the validity of the combinatorial map by verifying that condition discussed in Section 27.2.4 will be satisfied after the operation. This condition can be tested by using the method *is_sewable<i>(dh1,dh2)*. For example, the function *is_sewable<i>* would return *false* if we tried to sew a triangular facet with a quad facet. Note that given two darts *d1* and *d2*, if there is such a bijection, it is uniquely defined. So giving the two darts as arguments of the *sew<i>* is enough to retrieve all the pairs of darts to link. If such a bijection exists, the *sew<i>(dh1,dh2)* operation consists only in linking by β_i each couple of darts *d3* and *d4* such that $d3=f(d4)$.

In addition, the sew operation updates the associations between darts and non void attributes in order to guarantee that all the darts belonging to a given cell are associated with the same attribute (which is a condition of combinatorial map validity). For each couple of *j*-cells *c1* and *c2* that are merged into one *j*-cell during the sew, we have to update the two associated attributes *attr1* and *attr2*. If both are NULL, there is nothing to do. If one is NULL and the other not, we only associate the non NULL attribute to all the darts of the resulting cell. When the two attributes are non NULL, we first apply functor *On_merge* on the two attributes *attr1* and *attr2* (see Section 27.3.4). Then, we associate the attribute *attr1* to all darts of the resulting *j*-cell. Finally, attribute *attr2* is removed from the combinatorial map.

Note that when the two attributes are non NULL, the first one is kept. But user can customize this behavior in order to update the information contained in the attributes according to its needs. For that, we can define a specific functor, and use it as template argument for *OnMerge* parameter of the *Cell_attribute* definition. This functor can for example copy the information of the second attribute in the information of the first one to make as if the second attribute is kept.

For example, in Figure 27.10, we want to 3-sew the two initial volumes. *sew<3>(1,5)* links by β_3 the pairs of darts (1,5), (2,8), (3,7) and (4,6), thus the combinatorial map obtained is valid. 2-attributes are updated so that all the darts belonging to the 2-cell containing dart 1 become associated to the same 2-attribute after the operation.

Similarly, *unsew<i>(dh0)* operation unlinks β_i for all the darts in the orbit $\langle\beta_1,\ldots,\beta_{i-2},\beta_{i+2},\ldots,\beta_d\rangle(d0)$, and thus guarantees to obtain a valid combinatorial map. This operation is possible for any non *i*-free dart.

As for the sew operations, attributes are updated to guarantee that two darts belonging to two different *j*-cells are associated to two different *j*-attributes. If the unsew operation splits a *j*-cell *c* in two *j*-cells *c1* and *c2*, and if *c* is associated to a *j*-attribute *attr1*, then this attribute is duplicated into *attr2*, and all the darts belonging to *c2* are associated with this new attribute. Finally, we call the functor *On_split* on the two attributes *attr1* and *attr2* (see Section 27.3.4).

Let us consider the combinatorial map given in Figure 27.10 (Right). If we call *unsew<3>(2)*, we obtain the combinatorial map in Figure 27.10 (Left) (except for the color of the attribute associated to the 2-cell {5,6,7,8} which would be `#00ff00`). The *unsew<3>* operation has duplicated the 2-attribute associated to the 2-cell {1,2,3,4,5,6,7,8} since this 2-cell is split in two after the unsew operation.

advanced

If one wants to modify a combinatorial map *manually*, it is possible to switch off the updating between darts and attributes by passing *false* as last argument of *sew<i>(dh1,dh2,update_attributes=true)* and *unsew<i>(dh0,update_attributes=true)*. In these cases, the combinatorial map obtained may be no longer valid due to incorrect associations between darts and attributes.

In Figure 27.10 (Left), if we call *sew<3>(1,5,false)*, the resulting combinatorial map is similar to the combinatorial map of Figure 27.10 (Right) (we have linked by β_3 the pairs of darts (1,5), (2,8), (3,7) and (4,6)), but associations between darts and attributes are not valid. Indeed, we have kept the four initial attributes and all the associations between darts and attributes, thus two darts belonging to the same 2-cell (for example darts 1 and 5) are associated with two different attributes.

We can also use the *link_beta<i>(dh1,dh2,update_attributes=true)* which links *d1* and *d2* by β_i without modifying the other links. Association between darts and attributes are only modified for darts *d1* and *d2*, and similarly as for *sew<i>*, this updating can be avoided by passing *false* as last argument of *link_beta<i>(dh1,dh2,update_attributes)*. Lastly, we can use *unlink_beta<i>(dh0)* to unlink *d0* for β_i. In this last case, there is no modification of association between darts and attributes.

In Figure 27.10 (Left), if we call *link_beta<3>(1,5)*, in the resulting combinatorial map we have now $\beta_3(1)$=5 and $\beta_3(5)$=1. This combinatorial map is no longer valid (for example dart 2 is 3-free and we should have $\beta_3(2)$=8).

advanced

27.5.2 Removal and Insertion Operations

The following high level operations are defined as global functions taking an instance *cm* of *CombinatorialMap* as first argument. All these methods ensure that given a valid combinatorial map and a possible operation, the modified combinatorial map is also valid.

The first one is *remove_cell<CMap,i>(cm,dh0)* which modifies the combinatorial map to remove the *i*-cell containing dart *d0*, with $0 \leq i \leq d$. This operation is possible if *i=d* or if the given *i*-cell is incident to at most two *(i+1)*-cells which can be tested thanks to *is_removable<CMap,i>(cm,dh0)*. If the removed *i*-cell was incident to two different *(i+1)*-cells, these two cells are merged into one *(i+1)*-cell. In this case, the *On_merge* functor is called if two *(i+1)*-attributes are associated to the two *(i+1)*-cells. If the *i*-cell is associated with a non void attribute, it is removed from the combinatorial map (see three examples on Figures 27.11, 27.13 and 27.14).

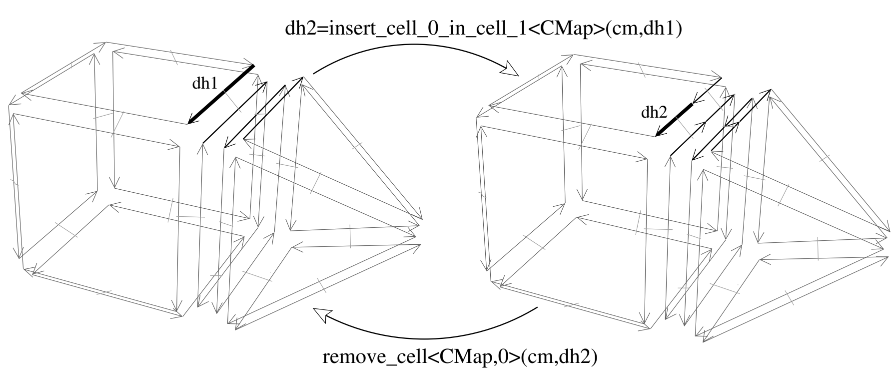


Figure 27.11: Example of *insert_cell_0_in_cell_1* and *remove_cell<0>* operations. **Left**: Initial combinatorial map. **Right**: After the insertion of a 0-cell in the 1-cell containing dart *d1*. Now if we remove the 0-cell containing dart *d2*, we obtain the initial combinatorial map.

The inverse operation of the removal is the insertion operation. Several versions exist, sharing a common principle. They consist in adding a new *i*-cell "inside" an existing *j*-cell, $i<j$, by splitting the *j*-cell into several *j*-cells. Contrary to *remove_cell<CMap,i>*, is it not possible to define a unique *insert_cell_i_in_cell_j<CMap,i,j>* function because parameters are different depending on *i* and *j*.

insert_cell_0_in_cell_1<CMap>(cm,dh0) adds a 0-cell in the 1-cell containing dart *d0*. The 1-cell is split in two. This operation is possible if *d0*$\in$*cm.darts()* (see example on Figure 27.11).

insert_cell_0_in_cell_2<CMap>(cm,dh0) adds a 0-cell in the 2-cell containing dart *d0*. The 2-cell is split in triangles, one for each initial edge of the facet. This operation is possible if *d0*$\in$*cm.darts()* (see example on Figure 27.12).

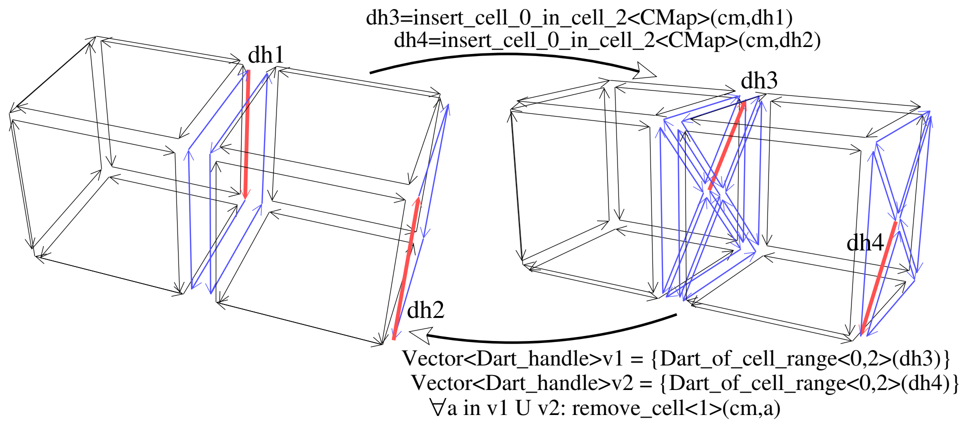


Figure 27.12: Example of *insert_cell_0_in_cell_2* operation.

insert_cell_1_in_cell_2<CMap>(cm,dh1,dh2) adds a 1-cell in the 2-cell containing darts *d1* and *d2*, between the two 0-cells containing darts *d1* and *d2*. The 2-cell is split in two. This operation is possible if $d1 \in \langle \beta_1 \rangle (d2)$ which can be tested thanks to *is_insertable_cell_1_in_cell_2(cm,dh1,dh2)*. In the example on Figure 27.13, it is possible to insert an edge between darts *d2* and *d3*, but it is not possible to insert an edge between *d1* and *d3*.

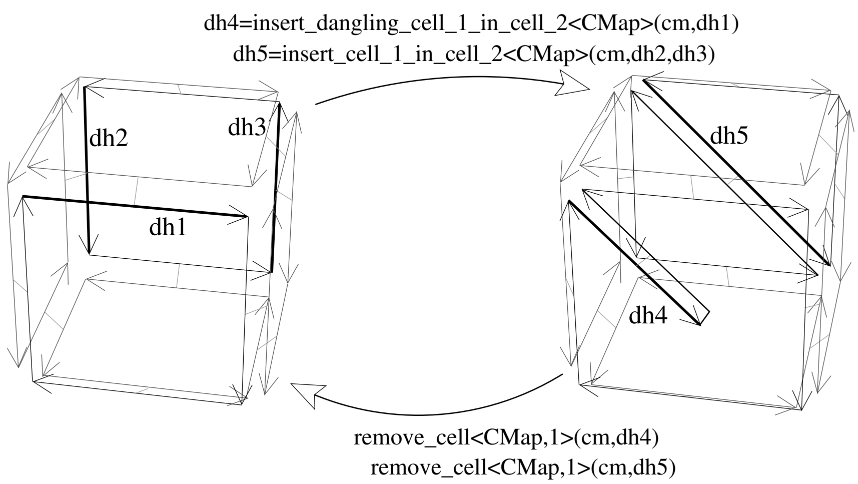


Figure 27.13: Example of *insert_cell_1_in_cell_2* and *remove_cell<1>* operations. **Left**: Initial combinatorial map. **Right**: After the insertion of two 1-cells: a first one between the two 0-cells containing darts *d2* and *d3*, and a second one incident to the 0-cell containing dart *d1*. Now if we remove the two 1-cells containing darts *d4* and *d5*, we obtain the initial combinatorial map.

insert_dangling_cell_1_in_cell_2<CMap>(cm,dh0) adds a 1-cell in the 2-cell containing dart *d0*, the 1-cell being attached by only one of its vertex to the 0-cell containing dart *d0*. This operation is possible if *d0∈cm.darts()*.

insert_cell_2_in_cell_3<CMap>(cm,itbegin,itend) adds a 2-cell in the 3-cell containing all the darts between *itbegin* and *itend*, along the path of 1-cells containing darts in [*itbegin*,*itend*). The 3-cell is split in two. This operation is possible if all the darts in [*itbegin*,*itend*) form a closed path inside a same 3-cell which can be tested thanks to *is_insertable_cell_2_in_cell_3(cm,itbegin,itend)* (see example on Figure 27.14).

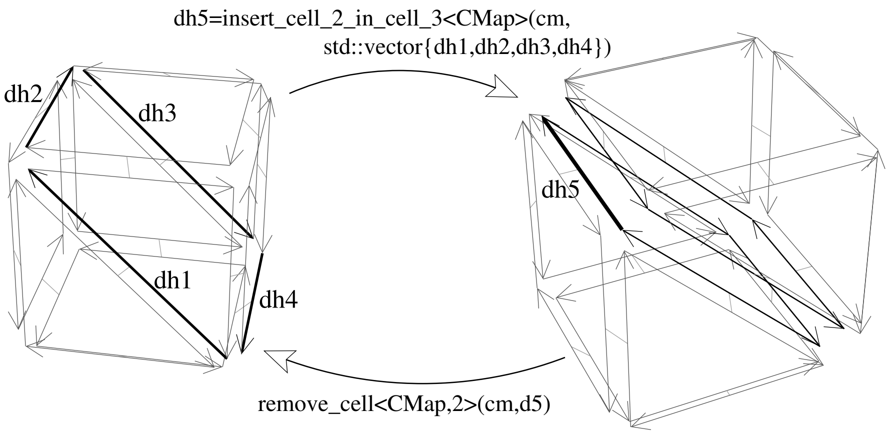


Figure 27.14: Example of *insert_cell_2_in_cell_3* and *remove_cell<2>* operations. **Left**: Initial combinatorial map. **Right**: After the insertion of a 2-cell along the path of 1-cells containing respectively *d1,d2,d3,d4*. Now if we remove the 2-cell containing dart *d5*, we obtain the initial combinatorial map.

Some examples of use of these operations are given in Section 27.6.2.

27.6 Examples

27.6.1 A 3D Combinatorial Map

In this example, a 3-dimensional combinatorial map is constructed. Two combinatorial tetrahedra are created, then the numbers of cells of the combinatorial map are displayed, and the validity of the combinatorial map is checked. Then, we illustrate the use of ranges to iterate over specific darts. The first loop enumerates all the darts of the first tetrahedron by using the range *Dart_of_orbit_range<1,2>*, and the second loop enumerates all the darts of the facet containing dart *dh2* by using the range *Dart_of_orbit_range<1>*.

```
#include <CGAL/Combinatorial_map.h>
#include <CGAL/Combinatorial_map_constructors.h>
#include <iostream>
#include <cstdlib>

typedef CGAL::Combinatorial_map<3> CMap_3;
typedef CMap_3::Dart_const_handle Dart_const_handle;

  int main()
  {
    CMap_3 cm;

    // Create two tetrahedra.
```

```
  Dart_const_handle dh1 = CGAL::make_combinatorial_tetrahedron(cm);
  Dart_const_handle dh2 = CGAL::make_combinatorial_tetrahedron(cm);

  // Display the combinatorial map characteristics.
  cm.display_characteristics(std::cout);
  std::cout<<", valid="<<cm.is_valid()<<std::endl;

  unsigned int res = 0;
  // Iterate over all the darts of the first tetrahedron.
  // Note that CMap_3::Dart_of_orbit_range<1,2> in 3D is equivalent to
  // CMap_3::Dart_of_cell_range<3>.
  for  (CMap_3::Dart_of_orbit_range<1,2>::const_iterator
          it(cm.darts_of_orbit<1,2>(dh1).begin()),
          itend(cm.darts_of_orbit<1,2>(dh1).end()); it!=itend; ++it)
    ++res;

  std::cout<<"Number of darts of the first tetrahedron: "
           <<res<<std::endl;

  res = 0;
  // Iterate over all the darts of the facet containing dh2.
  for (CMap_3::Dart_of_orbit_range<1>::const_iterator
         it(cm.darts_of_orbit<1>(dh2).begin()),
         itend(cm.darts_of_orbit<1>(dh2).end()); it!=itend; ++it)
    ++res;

  std::cout<<"Number of darts of the facet containing dh2: "
           <<res<<std::endl;

  return EXIT_SUCCESS;
}
```

File: examples/Combinatorial_map/map_3_simple_example.cpp

The output is:

```
#Darts=24, #0-cells=8, #1-cells=12, #2-cells=8, #3-cells=2, #ccs=2, valid=1
Number of darts of the first tetrahedron: 12
Number of darts of the facet containing dh2: 3
```

which gives the number of darts of the combinatorial map, the numbers of different cells, the number of connected components, and finally a Boolean showing the validity of the combinatorial map (a tetrahedron is made up of 12 darts because there are 3 darts per facet and there are 4 facets).

Note the creation in the for loops of the two instances of *Dart_of_orbit_range::const_iterator*: *it* is the current iterator, and *itend* an iterator to the end of the range. Having *itend* avoids calling *cm.darts_of_orbit<1,2> (dh1).end()* again and again as in the following example (which is a bad solution):

```
for (CMap_3::Dart_of_orbit_range<1,2>::const_iterator
       it(cm.darts_of_orbit<1,2>(dh1).begin());
       it!=cm.darts_of_orbit<1,2>(dh1).end(); ++it)
{...}
```

27.6.2 High Level Operations

This example shows some uses of high level operations. First we create a combinatorial hexahedron, the combinatorial map obtained is shown in Figure 27.15 (Left). Then we insert two 1-cells along two opposite 2-cells of the hexahedron. The combinatorial map obtained is shown in Figure 27.15 (Middle). Finally, we insert a 2-cell in the diagonal of the hexahedron in order to split it into two parts. We obtain the combinatorial map shown in Figure 27.15 (Right). We display the characteristics of the combinatorial map and check its validity.

The second part of this example removes the inserted elements. First we remove the inserted 2-cell, then the two inserted 1-cells. We get back the initial combinatorial hexahedron, which is verified by displaying once again the characteristics of the combinatorial map.

```
#include <CGAL/Combinatorial_map.h>
#include <CGAL/Combinatorial_map_constructors.h>
#include <CGAL/Combinatorial_map_operations.h>
#include <iostream>
#include <cstdlib>

typedef CGAL::Combinatorial_map<3> CMap_3;
typedef CMap_3::Dart_handle        Dart_handle;

int main()
{
  CMap_3 cm;

  // Create one combinatorial hexahedron.
  Dart_handle dh1 = CGAL::make_combinatorial_hexahedron(cm);

  // Add two edges along two opposite facets.
  CGAL_assertion( CGAL::is_insertable_cell_1_in_cell_2
                        (cm,dh1->beta(1),dh1->beta(0)) );

  CGAL::insert_cell_1_in_cell_2(cm,dh1->beta(1),dh1->beta(0));
  CGAL_assertion( cm.is_valid() );

  Dart_handle dh2=dh1->beta(2)->beta(1)->beta(1)->beta(2);

  CGAL_assertion( CGAL::is_insertable_cell_1_in_cell_2
                        (cm,dh2,dh2->beta(1)->beta(1)) );

  CGAL::insert_cell_1_in_cell_2(cm,dh2,dh2->beta(1)->beta(1));
  CGAL_assertion( cm.is_valid() );

  // Insert a facet along these two new edges plus two initial edges
  // of the hexahedron.
  std::vector<Dart_handle> path;
  path.push_back(dh1->beta(1));
  path.push_back(dh1->beta(0)->beta(2)->beta(1));
  path.push_back(dh2->beta(0));
  path.push_back(dh2->beta(2)->beta(1));

  CGAL_assertion( (CGAL::is_insertable_cell_2_in_cell_3
                   (cm,path.begin(),path.end())) );
```

```
  Dart_handle
   dh3=CGAL::insert_cell_2_in_cell_3(cm,path.begin(),path.end());
  CGAL_assertion( cm.is_valid() );

  // Display the combinatorial map characteristics.
  cm.display_characteristics(std::cout) << ", valid=" <<
    cm.is_valid() << std::endl;

  // We use the removal operations to get back to the initial hexahedron.
  CGAL_assertion( (CGAL::is_removable<CMap_3, 2>(cm,dh3)) );
  CGAL::remove_cell<CMap_3,2>(cm,dh3);
  CGAL_assertion( cm.is_valid() );

  CGAL_assertion( (CGAL::is_removable<CMap_3, 1>(cm,dh1->beta(1))) );
  CGAL::remove_cell<CMap_3,1>(cm,dh1->beta(1));
  CGAL_assertion( cm.is_valid() );

  CGAL_assertion( (CGAL::is_removable<CMap_3, 1>(cm,dh2->beta(0))) );
  CGAL::remove_cell<CMap_3,1>(cm,dh2->beta(0));
  CGAL_assertion( cm.is_valid() );

  // Display the combinatorial map characteristics.
  cm.display_characteristics(std::cout) << ", valid="
                                        << cm.is_valid() << std::endl;

  return EXIT_SUCCESS;
}
```

File: examples/Combinatorial_map/map_3_operations.cpp

The output is:

```
#Darts=36, #0-cells=8, #1-cells=14, #2-cells=9, #3-cells=2, #ccs=1, valid=1
#Darts=24, #0-cells=8, #1-cells=12, #2-cells=6, #3-cells=1, #ccs=1, valid=1
```

The first line gives the characteristics of the combinatorial map after all the insertions (the combinatorial map drawn in Figure 27.15 (Right)). There are two 3-cells (since the combinatorial hexahedron was split in two by the 2-cell insertion), nine 2-cells (since two 2-cells of the original hexahedron were split in two by the two 1-cell insertions, and a new 2-cell was created during the 2-cell insertion), fourteen 1-cells (since there are two new 1-cells created by the 1-cell insertion) while the number of 0-cells remains unchanged.

The second line is the result after the removal operations. We retrieve the original combinatorial hexahedron since we have removed all the inserted elements.

27.6.3 A 4D Combinatorial Map

In this example, a 4-dimensional combinatorial map is used. Two tetrahedral cells are created and sewn by β_4. Then the numbers of cells of the combinatorial map are displayed, and its validity is checked.

By looking at these numbers of cells, we can see that the 4D combinatorial map contains only one 3-cell. Indeed, the *sew<4>* operation has identified by pairs all the darts of the two 3-cells by definition of the sew operation

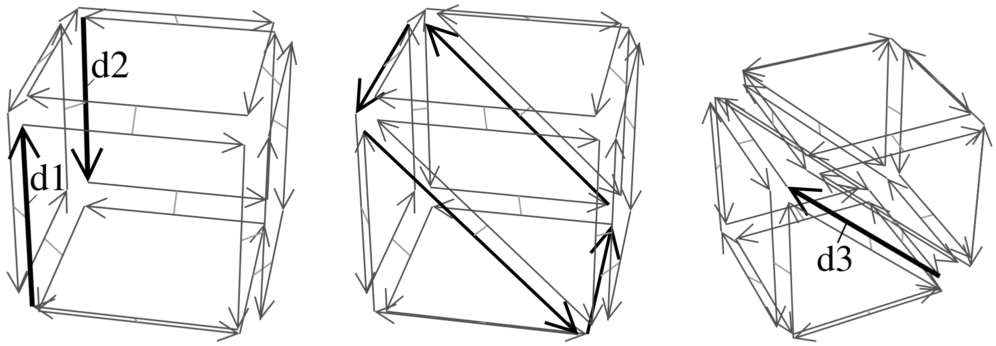


Figure 27.15: Example of high level operations. **Left**: Initial 3D combinatorial map after the creation of the combinatorial hexahedron. **Middle**: Combinatorial map obtained after the two 1-cell insertions. The two 2-cells were split in two. **Right**: Combinatorial map obtained after the 2-cell insertion. The 3-cell was split in two.

(see Section 27.5.1) which, in 4D, links by β_3 all the darts in $\langle\beta_1,\beta_2\rangle(d1)$ and in $\langle\beta_1,\beta_2\rangle(d2)$. The situation is similar (but in higher dimension) to the configuration where we have two triangles in a 3D combinatorial map, and you use *sew<3>* between these two triangles. The two triangles are identified since all their darts are linked by β_3, thus we obtain a 3D combinatorial map containing only one 3-cell. Note that this 3-cell is unbounded since the darts of the two triangles are all 2-free. In the 4D case, the 4-cell is unbounded since all its darts are 3-free.

In this example, we also illustrate how to use the basic methods to build "by hand" some specific configuration in a combinatorial map. In fact, these functions are already present in the package: function *make_triangle(cm)* is equivalent to *CGAL::make_combinatorial_polygon(cm,3)* and *make_tetrahedron(cm)* is equivalent to *CGAL::make_combinatorial_tetrahedron(cm)*. If we want to create a 4D simplex, we must create five 3D simplexes, and sew them correctly two by two by β_3 (and so on if you want to create higher dimensional combinatorial map).

```
#include <CGAL/Combinatorial_map.h>
#include <CGAL/Combinatorial_map_constructors.h>
#include <iostream>
#include <cstdlib>

typedef CGAL::Combinatorial_map<4> CMap_4;
typedef CMap_4::Dart_handle Dart_handle;

Dart_handle make_triangle(CMap_4& amap)
{
 Dart_handle d1 = amap.create_dart();
 Dart_handle d2 = amap.create_dart();
 Dart_handle d3 = amap.create_dart();
 amap.link_beta<1>(d1,d2);
 amap.link_beta<1>(d2,d3);
 amap.link_beta<1>(d3,d1);
 return d1;
}

Dart_handle make_tetrahedral(CMap_4& amap)
{
  Dart_handle d1 = make_triangle(amap);
  Dart_handle d2 = make_triangle(amap);
  Dart_handle d3 = make_triangle(amap);
```

```
  Dart_handle d4 = make_triangle(amap);
  amap.link_beta<2>(d1, d2);
  amap.link_beta<2>(d3, d2->beta(0));
  amap.link_beta<2>(d1->beta(1), d3->beta(0));
  amap.link_beta<2>(d4, d2->beta(1));
  amap.link_beta<2>(d4->beta(0), d3->beta(1));
  amap.link_beta<2>(d4->beta(1), d1->beta(0));
  return d1;
}

int main()
{
  CMap_4 cm;
  Dart_handle d1 = make_tetrahedral(cm);
  Dart_handle d2 = make_tetrahedral(cm);

  cm.sew<4>(d1,d2);

  cm.display_characteristics(std::cout);
  std::cout<<", valid="<<cm.is_valid()<<std::endl;

  return EXIT_SUCCESS;
}
```

File: examples/Combinatorial_map/map_4_simple_example.cpp

The output is:

```
#Darts=24, #0-cells=4, #1-cells=6, #2-cells=4, #3-cells=1, #4-cells=2, #ccs=1, valid=1
```

27.6.4 Combinatorial Map With Attributes

In the following example, we illustrate how to specify the 2-attributes in a 3D combinatorial map. For that, we define our own item class using *CGAL::Dart<3,CMap>* as type of dart, and *CGAL::Cell_attribute< CMap,int,CGAL::Tag_true,Sum_functor,Divide_by_two_functor>* as attributes which contain an *int* and which are associated to 2-cells of the combinatorial map.

Functors *Sum_functor* and *Divide_by_two_functor* define a custom behavior: when two attributes *ca1* and *ca2* are merged into *ca1*, the value of *ca1* is the sum of the two initial values. When an attribute *ca1* is split in the two attributes *ca1* and *ca2*, the value of each attribute is half of the first value.

```
#include <CGAL/Combinatorial_map.h>
#include <CGAL/Combinatorial_map_constructors.h>
#include <CGAL/Combinatorial_map_operations.h>
#include <CGAL/Cell_attribute.h>
#include <iostream>
#include <algorithm>
#include <cstdlib>

struct Sum_functor
{
```

```
  template<class Cell_attribute>
  void operator()(Cell_attribute& ca1,Cell_attribute& ca2)
  { ca1.info()=ca1.info()+ca2.info(); }
};
struct Divide_by_two_functor
{
  template<class Cell_attribute>
  void operator()(Cell_attribute& ca1,Cell_attribute& ca2)
  {
    ca1.info()=(ca1.info()/2);
    ca2.info()=(ca1.info());
  }
};

struct Myitem
{
  template<class CMap>
  struct Dart_wrapper
  {
    typedef CGAL::Dart<3, CMap> Dart;
    typedef CGAL::Cell_attribute<CMap, int, CGAL::Tag_true,
                                 Sum_functor, Divide_by_two_functor>
    Facet_attribute;
    typedef CGAL::cpp0x::tuple<void,void,Facet_attribute> Attributes;
  };
};

typedef CGAL::Combinatorial_map<3,Myitem> CMap_3;
typedef CMap_3::Dart_handle               Dart_handle;

int main()
{
  CMap_3 cm;

  // Create 2 hexahedra.
  Dart_handle dh1 = make_combinatorial_hexahedron(cm);
  Dart_handle dh2 = make_combinatorial_hexahedron(cm);

  // 1) Create all 2-attributes and associated them to darts.
  for (CMap_3::Dart_range::iterator
         it=cm.darts().begin(), itend=cm.darts().end();
         it!=itend; ++it)
    {
      if ( it->attribute<2>()==NULL )
        cm.set_attribute<2>(it, cm.create_attribute<2>());
    }

  // 2) Set the color of all facets of the first hexahedron to 7.
  for (CMap_3::One_dart_per_incident_cell_range<2, 3>::iterator
         it=cm.one_dart_per_incident_cell<2,3>(dh1).begin(),
         itend=cm.one_dart_per_incident_cell<2,3>(dh1).end(); it!=itend;
   ++it)
    { it->attribute<2>()->info()=7; }
```

```
  // 3) Set the color of all facets of the second hexahedron to 13.
  for (CMap_3::One_dart_per_incident_cell_range<2, 3>::iterator it=
         cm.one_dart_per_incident_cell<2,3>(dh2).begin(),
         itend=cm.one_dart_per_incident_cell<2,3>(dh2).end(); it!=itend;
   ++it)
    { it->attribute<2>()->info()=13; }

  // 4) 3-Sew the two hexahedra along one facet.
  cm.sew<3>(dh1, dh2);

  // 5) Display all the values of 2-attributes.
  for (CMap_3::Attribute_range<2>::type::iterator
         it=cm.attributes<2>().begin(), itend=cm.attributes<2>().end();
       it!=itend; ++it)
    {
      std::cout<<it->info()<<"; ";
    }
  std::cout<<std::endl;

  // 6) Insert a vertex in the facet between the two hexahedra.
  CGAL::insert_cell_0_in_cell_2(cm, dh2);

  // 7) Display all the values of 2-attributes.
  for (CMap_3::Attribute_range<2>::type::iterator
         it=cm.attributes<2>().begin(), itend=cm.attributes<2>().end();
       it!=itend; ++it)
    {
      std::cout<<it->info()<<"; ";
    }
  std::cout<<std::endl;
  cm.display_characteristics(std::cout);
  std::cout<<", valid="<<cm.is_valid()<<std::endl;

  return EXIT_SUCCESS;
}
```

File: examples/Combinatorial_map/map_3_with_colored_facets.cpp

The output is:

```
20; 7; 7; 7; 7; 7; 13; 13; 13; 13; 13;
2; 7; 7; 7; 7; 7; 2; 13; 13; 13; 13; 13; 5; 10;
#Darts=64, #0-cells=13, #1-cells=24, #2-cells=14, #3-cells=2, #ccs=1, valid=1
```

Before the *cm.sew<3>*, each 2-cell of the first cube is associated with an attribute having 7 as value, and each 2-cell of the second cube with an attribute having 13 as value. During the *cm.sew<3>*, two 2-cells are merged, thus the functor *Sum_functor* is called on the two associated 2-attributes, and the value of the new 2-cell is the sum of the two previous one: 20.

Then we call *CGAL::insert_cell_0_in_cell_2* on a dart which belong to this 2-cell. This method splits the existing 2-cell in *k* 2-cells, *k* being the number of 1-cells of the initial 2-cell (4 in this example). These splits are made consecutively, thus for the first split, we create a new attribute as copy of the initial one and call functor *Divide_by_two_functor* on these two attributes: the value of each attribute is thus 20/2=10. For the second split, the value of each attribute is thus 10/2=5, and for the last split the value of each attribute is thus 5/2=2 (remember that information contained in 2-attributes in an *int*). At the end, we obtain five 2-attributes with 7 as value, five 2-attributes with 13 as value, and four 2-attributes having respectively 2, 2, 5 and 10 as values.

27.7 Mathematical Definitions

The initial definition of combinatorial map in any dimension is given in [Lie91, Lie94]. But it allows only to represent objects without boundaries. This definition was extended [PABL07, Dam10] in order to allow to represent objects with boundaries, based on the notions of partial permutations and partial involutions.

Intuitively, a *partial permutation* on a finite set E is a mapping from $E \cup \{\varnothing\}$ to $E \cup \{\varnothing\}$ which is injective on the subset of the domain that does not map to $\varnothing$. More precisely, a mapping p: $E \cup \{\varnothing\} \rightarrow E \cup \{\varnothing\}$ is a *partial permutation* defined on E if:

1. $p(\varnothing)=\varnothing$;
2. $\forall e1 \in E$, $\forall e2 \in E$, $p(e1)=p(e2) \neq \varnothing \Rightarrow e1=e2$.

The inverse p^{-1} of this partial permutation is also a partial permutation and is defined by:

1. $p^{-1}(\varnothing)=\varnothing$;
2. $\forall e \in E$, if it exists $a \in E$ such that $p(a)=e$, then $p^{-1}(e)$=a, otherwise $p^{-1}(e)=\varnothing$.

Let E be a set, and p a partial permutation on E. An element e is called a *fixed point* for p if $p(e)=e$. p is a *partial involution* if $\forall e \in E$: $p(e) \neq \varnothing \Rightarrow p(p(e))=e$.

Now we can give the definition of a combinatorial map in any dimension. Let $d \geq 0$. A d-dimensional combinatorial map (or d-map) is a (d+1)-tuple $M=(D,\beta_1,\ldots,\beta_d)$ where:

1. D is a finite set of darts;
2. β_1 is a partial permutation on D;
3. $\forall i$, $2 \leq i \leq d$, β_i is a partial involution on D without fixed point;
4. $\forall i$: $0 \leq i \leq d$-2, $\forall j$: $3 \leq j \leq d$, $i+2 \leq j$, $\beta_i \circ \beta_j$ is a partial involution.

A d-dimensional combinatorial map represents a subdivision of an orientable d-dimensional quasi-manifold. A dart is an abstract element which is only required to define partial permutations. The last line of the definition fixes constraints which guarantee the topological validity of the represented object, i.e., the fact that it is a quasi-manifold. This definition allows us to verify the validity of a given combinatorial map by checking if each item of the definition is satisfied.

Given a set of partial permutations $S=\{f_1,\ldots,f_k\}$, we denote by $\langle S \rangle$ the *permutation group* generated by $\{f_1,\ldots,f_k\}$ and whose group operation is the composition of partial permutations. The orbit $\langle f_1,\ldots,f_k \rangle(a)$ is the set of darts which can be obtained from a by elements of $\langle S \rangle$: $\langle f_1,\ldots,f_k \rangle(a)=\{\phi(a) | \phi \in \langle S \rangle\} \setminus \{\varnothing\}$.

Let $d0 \in D$ be a dart. Given i, $1 \leq i \leq d$, the i-cell containing $d0$ is $\langle \beta_1,\ldots,\beta_{i-1},\beta_{i+1},\ldots,\beta_d \rangle(d0)$. The 0-cell containing $d0$ is $\langle \{\beta_i \circ \beta_j | \forall i,j$: $1 \leq i < j \leq d\} \rangle(d0)$.

27.8 Design and Implementation History

The code of this package is inspired by Moka, a 3D topological modeler mainly developed by Frédéric Vidil and Guillaume Damiand (http://moka-modeller.sourceforge.net/). However, Moka was based on Generalized maps (and not Combinatorial maps), and the design was not CGAL "compatible". Thus, Guillaume Damiand started to develop a totally new package by mixing ideas taken from Moka with the design of the Halfedge data structure package of CGAL. Andreas Fabri and Sébastien Loriot contributed to the design, the coding, and to the documentation of the package, and Laurent Rineau helped for the design. Emma Michel contributed to the manual. Monique Teillaud and Bernd Gärtner contributed to the manual by giving useful remarks, really numerous and detailed for Monique.

Combinatorial map Reference Manual

27.9 Classified Reference Pages

27.9.1 Concepts

27.9.2 Classes

27.9.3 Global Functions

Constructions for Combinatorial Maps

Operations for Combinatorial Maps

27.10 Alphabetical List of Reference Pages

CombinatorialMap

Definition

The concept CombinatorialMap defines a d-dimensional combinatorial map.

Creation

CombinatorialMap cm;	Default constructor.

Types

CombinatorialMap:: Dart	Dart type, a model of the *Dart* concept.
CombinatorialMap:: Dart_handle	Dart handle type, equal to *Dart::Dart_handle*.
CombinatorialMap:: Dart_const_handle	Dart const handle type, equal to *Dart::Dart_const_handle*.
CombinatorialMap:: size_type	Size type (an unsigned integral type).

Constants

static unsigned int	*dimension;*	The dimension d of *cm*, equal to *Dart::dimension.*
static size_type	*NB_MARKS;*	The number of available Boolean marks of *cm*.
static Dart_handle	*null_dart_handle;*	The null dart handle constant: a dart *d0* is *i*-free if *d0.beta(i)==null_dart_handle*. Note that **null_dart_handle*$\notin$*cm.darts()*.

Types for attributes

CombinatorialMap:: Attributes

The tuple of attributes, containing at most *dimension+1* types (one for each possible cell of the combinatorial map). Each type of the tuple must be either a model of the *CellAttribute* concept or *void*. The first type corresponds to 0-attributes, the second to 1-attributes and so on. If the i^{th} type in the tuple is *void*, *(i-1)*-attributes are disabled. Otherwise, *(i-1)*-attributes are enabled and have the given type. If the size of the tuple is k, with k<*dimension+1*, $\forall i$: $k \leq i \leq$ *dimension*, i-attributes are disabled.

CombinatorialMap:: template<unsigned int i> Attribute_type<i>::type

Type of i-attributes, a model of *CellAttribute* concept. *Attribute_type<i>::type::Dart_handle* is equal to *Dart_handle*, and *Attribute_type<i>::type::Dart_const_handle* is equal to *Dart_const_handle*.
Precondition: $0 \leq i \leq$ *dimension* and i-attributes are non void.

CombinatorialMap:: template<unsigned int i> Attribute_handle<i>::type

Handle to i-attributes, equal to *Dart::Attribute_handle<i>::type*.
Precondition: $0 \leq i \leq$ *dimension* and i-attributes are non void.

CombinatorialMap:: template<unsigned int i> Attribute_const_handle<i>::type

Const handle to i-attributes, equal to *Dart::Attribute_const_handle<i>::type*.
Precondition: $0 \leq i \leq dimension$ and i-attributes are non void.

Range types

CombinatorialMap:: Dart_range

Range of all the darts of *cm*. This type is a model of *Range* concept, its iterator type is bidirectional and its value type is *Dart*.

CombinatorialMap:: Dart_const_range

Const range of all the darts of *cm*. This type is a model of *ConstRange* concept, its iterator type is bidirectional and its value type is *Dart*.

CombinatorialMap:: template<unsigned int i> Attribute_range<i>::type

Range of all the i-attributes (which must be non void), with $0 \leq i \leq dimension$. This type is a model of *Range* concept, its iterator type is bidirectional and its value type is *Attribute_type<i>::type*.

CombinatorialMap:: template<unsigned int i> Attribute_const_range<i>::type

Const range of all the i-attributes (which must be non void), with $0 \leq i \leq dimension$. This type is a model of *ConstRange* concept, its iterator type is bidirectional and its value type is *Attribute_type<i>::type*.

CombinatorialMap:: template<unsigned int ... Beta> Dart_of_orbit_range

Range of all the darts of the <*Beta...*> orbit. This type is a model of *Range* concept, its iterator type is forward and its value type is *Dart*.

CombinatorialMap:: template<unsigned int ... Beta> Dart_of_orbit_const_range

Const range of all the darts of the <*Beta...*> orbit. This type is a model of *ConstRange* concept, its iterator type is forward and its value type is *Dart*.

CombinatorialMap:: template<unsigned int i,unsigned int dim=dimension> Dart_of_cell_range

Range of all the darts of an i-cell in *dim* dimension (with $0 \leq i \leq dim+1$ and $0 \leq dim \leq dimension$). If $i==dim+1$, range of all the darts of a connected component. This type is a model of *Range* concept, its iterator type is forward and its value type is *Dart*.

CombinatorialMap:: template<unsigned int i,unsigned int dim=dimension> Dart_of_cell_const_range

Const range of all the darts of the i-cell in *dim* dimension (with $0 \leq i \leq dim+1$ and $0 \leq dim \leq dimension$). If $i==dim+1$, range of all the darts of a connected component. This type is a model of *ConstRange* concept, its iterator type is forward and its value type is *Dart*.

CombinatorialMap:: template<unsigned int i,unsigned int j,unsigned int dim=dimension> One_dart_per_incident_cell_range

Range of one dart of each *i*-cell incident to one *j*-cell. Cells are considered in *dim* dimension (with $0 \leq i \leq dim+1$, $0 \leq j \leq dim+1$ and $0 \leq dim \leq dimension$). If *i==dim+1*, consider each connected component instead of each *i*-cell. If *j==dim+1*, consider one connected component instead of one *j*-cell. This type is a model of *Range* concept, its iterator type is forward and its value type is *Dart*.

CombinatorialMap:: template<unsigned int i,unsigned int j,unsigned int dim=dimension> One_dart_per_incident_cell_const_range

Const range of one dart of each *i*-cell incident to one *j*-cell. Cells are considered in *dim* dimension (with $0 \leq i \leq dim+1$, $0 \leq j \leq dim+1$ and $0 \leq dim \leq dimension$). If *i==dim+1*, consider each connected component instead of each *i*-cell. If *j==dim+1*, consider one connected component instead of one *j*-cell. This type is a model of *ConstRange* concept, its iterator type is forward and its value type is *Dart*.

CombinatorialMap:: template<unsigned int i,unsigned int dim=dimension> One_dart_per_cell_range

Range of one dart of each *i*-cell of *cm*. Cells are considered in *dim* dimension (with $0 \leq i \leq dim+1$ and $0 \leq dim \leq dimension$). If *i==dim+1*, consider each connected component instead of each *i*-cell. This type is a model of *Range* concept, its iterator type is forward and its value type is *Dart*.

CombinatorialMap:: template<unsigned int i,unsigned int dim=dimension> One_dart_per_cell_const_range

Const range of one dart of each *i*-cell of *cm*. Cells are considered in *dim* dimension (with $0 \leq i \leq dim+1$ and $0 \leq dim \leq dimension$). If *i==dim+1*, consider each connected component instead of each *i*-cell. This type is a model of *ConstRange* concept, its iterator type is forward and its value type is *Dart*.

Access Member Functions

bool	*cm.is_empty() const*	Returns true iff *cm* is empty, i.e. it contains no dart.
bool	*cm.is_valid() const*	Returns true iff *cm* is valid.

A combinatorial map *cm* is valid (see Sections 27.2.1 and 27.2.4) if for all dart handle *dh* such that **dh∈cm.darts()*:

- *dh->is_free(0)*, or *dh->beta(0)->beta(1)==dh*;
- *dh->is_free(1)*, or *dh->beta(1)->beta(0)==dh*;

- $\forall i$, $2 \leq i \leq dimension$: *dh->is_free(i)*, or *dh->beta(i)->beta(i)==dh*;
- $\forall i,j$, $0 \leq i < i+2 \leq j \leq dimension$ such that $j \geq 3$: *dh->beta(i)->beta(j)==null_dart_handle* or *dh->beta(i)->beta(j)->beta(i)->beta(j)==dh*;
- $\forall i$, $0 \leq i \leq dimension$ such that *i*-attributes are non void: $\forall$*dh2* such that **dh2* belong to the same *i*-cell than **dh*: *dh2->attribute<i>()==dh->attribute<i>()*.

bool *cm.is_without_boundary(unsigned int i) const*

Returns true iff *cm* is wihout *i*-boundary (i.e. there is no *i*-free dart).
Precondition: $1 \leq i \leq dimension$.

bool *cm.is_without_boundary() const*

Returns true iff *cm* is without boundary in all dimensions.

size_type *cm.number_of_darts() const*

Returns the number of darts in *cm*.

template <unsigned int i>
size_type *cm.number_of_attributes() const*

Returns the number of *i*-attributes in *cm*.
Precondition: $0 \leq i \leq dimension$, and *i*-attributes are non void.

Dart_handle *cm.dart_handle(Dart& adart)*

Returns the dart handle of *adart*.
Precondition: *adart*$\in$*cm.darts()*.

Dart_const_handle *cm.dart_handle(Dart adart) const*

Returns the dart const handle of *adart*.
Precondition: *adart*$\in$*cm.darts()*.

Dart_handle *cm.beta(Dart_handle dh, int i, int j) const*

Returns $\beta_j(\beta_i(dh))$. Overloads of this member function are defined that take from one to nine integer as arguments. For each function, betas are applied in the same order as their indices are given as parameters. For example *beta(dh,1)*=$\beta_1(dh)$, and *beta(dh,1,2,3,0)*=$\beta_0(\beta_3(\beta_2(\beta_1(dh))))$.
Precondition: $0 \leq i \leq dimension$, $0 \leq j \leq dimension$ and **dh*$\in$*cm.darts()*.

Dart_const_handle *cm.beta(Dart_const_handle dh, int i, int j) const*

Returns $\beta_j(\beta_i(dh))$. Overloads of this member function are defined that take from one to nine integer as arguments.
Precondition: $0 \leq i \leq dimension$, $0 \leq j \leq dimension$ and **dh*$\in$*cm.darts()*.

template <unsigned int i>
bool *cm.is_sewable(Dart_const_handle dh1, Dart_const_handle dh2) const*

Returns true iff *dh1* can be *i*-sewn with *dh2* by keeping *cm* valid, i.e. if there is a bijection f between all the darts of the orbit $D1=\langle\beta_1,\ldots,\beta_{i-2},\beta_{i+2},\ldots,\beta_d\rangle(dh1)$ and $D2=\langle\beta_1,\ldots,\beta_{i-2},\beta_{i+2},\ldots,\beta_d\rangle(dh2)$ satisfying: $f(dh1)=dh2$, and for all $e\in D1$, for all $j\in\{1,\ldots,i\text{-}2,i\text{+}2,\ldots,d\}$, $f(\beta_j(e))=\beta_j^{-1}(f(e))$.
Precondition: $0\leq i\leq dimension$, $*dh1\in cm.darts()$, and $*dh2\in cm.darts()$.

std::ostream& *cm.display_characteristics(std::ostream & os) const*

Displays on *os* the characteristics of *cm*: its number of darts, its number of *i*-cells, for each *i*, $0\leq i\leq$CMap::*dimension*, and its number of connected components.

Example of output for a 3D combinatorial map containing two disjoint combinatorial tetrahedra:
`#Darts=24, #0-cells=8, #1-cells=12, #2-cells=8, #3-cells=2, #ccs=2`

Range Access Member Functions

Dart_range& *cm.darts()* Returns a range of all the darts in *cm*.
Dart_const_range& *cm.darts() const* Returns a const range of all the darts in *cm*.

template<unsigned int i>
Attribute_range<i>::type&

cm.attributes() Returns a range of all the *i*-attributes in *cm*.
Precondition: $0\leq i\leq dimension$, and *i*-attributes are non void.

template<unsigned int i>
Attribute_const_range<i>::type&

cm.attributes() const Returns a const range of all the *i*-attributes in *cm*.
Precondition: $0\leq i\leq dimension$, and *i*-attributes are non void.

template<unsigned int ... Beta>
Dart_of_orbit_range *cm.darts_of_orbit(Dart_handle dh)*

Returns a range of all the darts of the orbit *<Beta...>(dh)*.
Precondition: **dh*$\in$*cm.darts()* and *Beta...* is a sequence of integers $i_1,\ldots,i_k$, such that $0\leq i_1<i_2<\ldots<i_k\leq dimension$, and ($i_1\neq 0$ or $i_2\neq 1$).

template<unsigned int ... Beta>
Dart_of_orbit_const_range

cm.darts_of_orbit(Dart_const_handle dh) const

Returns a const range of all the darts of the orbit *<Beta...> (dh)*.
Precondition: Same as for the non const version.

template<unsigned int i,unsigned int dim=dimension>
Dart_of_cell_range *cm.darts_of_cell(Dart_handle dh)*

Returns a range of all the darts of the *i*-cell containing *dh*. *i*-cells are considered in *dim* dimension. If *i==dim+1*, range of all the darts of the connected component containing *dh*.
Precondition: **dh*$\in$*cm.darts()*, $0 \leq i \leq dim+1$ and $0 \leq dim \leq dimension$.

template<unsigned int i,unsigned int dim=dimension>
Dart_of_cell_const_range

cm.darts_of_cell(Dart_const_handle dh) const

Returns a const range of all the darts of the *i*-cell containing *dh*. *i*-cells are considered in *dim* dimension. If *i==dim+1*, const range of all the darts of the connected component containing *dh*.
Precondition: Same as for the non const version.

template<unsigned int i,unsigned int j,unsigned int dim=dimension>
One_dart_per_incident_cell_range

cm.one_dart_per_incident_cell(Dart_handle dh)

Returns a range of one dart of each *i*-cell incident to the *j*-cell containing *dh*. Cells are considered in *dim* dimension. If *i==dim+1*, consider each connected component instead of each *i*-cell. If *j==dim+1*, consider the connected component containing *dh* instead of the *j*-cell.
Precondition: **dh*$\in$*cm.darts()*, $0 \leq i \leq dim+1$, $0 \leq j \leq dim+1$ and $0 \leq dim \leq dimension$.

template<unsigned int i,unsigned int j,unsigned int dim=dimension>
One_dart_per_incident_cell_const_range

cm.one_dart_per_incident_cell(Dart_const_handle dh) const

Returns a const range of one dart of each *i*-cell incident to the *j*-cell containing *dh*. Cells are considered in *dim* dimension. If *i==dim+1*, consider each connected component instead of each *i*-cell. If *j==dim+1*, consider the connected component containing *dh* instead of the *j*-cell.
Precondition: Same as for the non const version.

template<unsigned int i,unsigned int dim=dimension>
One_dart_per_cell_range

cm.one_dart_per_cell()

Returns a range of one dart of each *i*-cell in *cm*. Cells are considered in *dim* dimension. If *i==dim+1*, range of one dart of each connected component in *cm*.
Precondition: $0 \leq i \leq dim+1$ and $0 \leq dim \leq dimension$.

template<unsigned int i,unsigned int dim=dimension>

One_dart_per_cell_const_range

cm.one_dart_per_cell() const

Returns a const range of one dart of each *i*-cell in *cm*. Cells are considered in *dim* dimension. If *i==dim+1*, const range of one dart of each connected component in *cm*.
Precondition: Same as for the non const version.

Modifiers

template<typename T1>
Dart_handle *cm.create_dart(T1 t1)*

Creates a new dart in *cm*, and returns the corresponding handle. Calls the constructor of dart having *T1* as parameter. A new dart is initialized to be *i*-free, $\forall i$: $0 \leq i \leq dimension$, and to have no associated attribute for each non void attribute. Overloads of this member function are defined that take from zero to nine arguments. With zero argument, *create_dart()* creates a new dart by using the default constructor.

void *cm.erase_dart(Dart_handle dh)*

Removes **dh* from *cm*.
Precondition: **dh*$\in$*cm.darts()*.

template<unsigned int i,typename T1>
Attribute_handle<i>::type

cm.create_attribute(T1 t1)

Creates a new *i*-attribute in *cm*, and returns the corresponding handle. Calls the constructor of *i*-attribute having *T1* as parameter. Overloads of this member function are defined that take from zero to nine arguments. With zero argument, *create_attribute<i>()* creates a new *i*-attribute by using the default constructor.
Precondition: $0 \leq i \leq dimension$, and *i*-attributes are non void.

template <unsigned int i>
void *cm.erase_attribute(Attribute_handle<i>::type ah)*

Removes the *i*-attribute **ah* from *cm*.
Precondition: $0 \leq i \leq dimension$, *i*-attributes are non void, and **ah*$\in$*cm.attributes<i>()*.

template <unsigned int i>

void *cm.set_attribute(Dart_handle dh, Attribute_handle<i>::type ah)*

Associates the i-attribute of all the darts of the i-cell containing *dh* to *ah*.
Precondition: **dh*$\in$*cm.darts()*, $0 \leq i \leq$ *dimension*, i-attributes are non void, and **ah*$\in$*cm.attributes<i>()*.

void *cm.clear()* Deletes all the darts and all the attributes of *cm*.

Operations

template <unsigned int i>
void *cm.sew(Dart_handle dh1, Dart_handle dh2, bool update_attributes=true)*

Links by β_i two by two all the darts of the orbit $D1=\langle\beta_1,\ldots,\beta_{i-2},\beta_{i+2},\ldots,\beta_d\rangle(dh1)$ and $D2=\langle\beta_0,\beta_2,\ldots,\beta_{i-2},\beta_{i+2},\ldots,\beta_d\rangle\ (dh2)$ such that $d2=f(d1)$, f being the bijection between $D1$ and $D2$ satisfying: $f(dh1)=dh2$, and for all $e \in D1$, for all $j \in \{1,\ldots,i\text{-}2,i\text{+}2,\ldots,d\}$, $f(\beta_j(e))=\beta_j^{-1}(f(e))$.
If *update_attributes* is *true*, when necessary, non void attributes are updated to ensure the validity of *cm*: for each j-cells *c1* and *c2* which are merged into one j-cell during the sew, the two associated attributes *attr1* and *attr2* are considered. If one attribute is NULL and the other not, the non NULL attribute is associated to all the darts of the resulting cell. When the two attributes are non NULL, functor *Attribute_type<i>::type::On_merge* is called on the two attributes *attr1* and *attr2*. Then, the attribute *attr1* is associated to all darts of the resulting j-cell. Finally, attribute *attr2* is removed from *cm*.
Precondition: *cm.is_sewable<i>(dh1,dh2)*.

advanced

If *update_attributes* is *false*, non void attributes are not updated; thus *cm* can be no more valid after this modification.

advanced

template <unsigned int i>

void *cm.unsew(Dart_handle dh, bool update_attributes=true)*

Unlinks by β_i all the darts in the orbit $\langle\beta_1,\ldots,\beta_{i-2},\beta_{i+2},\ldots,\beta_d\rangle(dh)$. If *update_attributes* is *true*, when necessary, non void attributes are updated to ensure the validity of *cm*: for each *j*-cell *c* split in two *j*-cells *c1* and *c2* by the operation, if *c* is associated to a *j*-attribute *attr1*, then this attribute is duplicated into *attr2*, and all the darts belonging to *c2* are associated with this new attribute. Finally, the functor *Attribute_type<i>::type::On_split* is called on the two attributes *attr1* and *attr2*.
Precondition: $0 \leq i \leq dimension$, $*dh \in cm.darts()$ and *dh* is not *i*-free.

advanced

If *update_attributes* is *false*, non void attributes are not updated thus *cm* can be no more valid after this modification.

advanced

advanced

template <unsigned int i>
void *cm.link_beta(Dart_handle dh1, Dart_handle dh2, bool update_attributes=true)*

Links *dh1* and *dh2* by β_i. *cm* can be no more valid after this modification. If *update_attributes* is true, non void attributes of *dh1* and *dh2* are updated: if one dart has an attribute and the second dart not, the non null attribute is associated to the dart having a null attribute. If both darts have an attribute, the attribute of *dh1* is associated to *dh2*.
Precondition: $0 \leq i \leq dimension$, $*dh1 \in cm.darts()$, $*dh2 \in cm.darts()$ and ($i<2$ or $dh1 \neq dh2$).

template <unsigned int i>
void *cm.unlink_beta(Dart_handle dh)*

Unlinks *dh* and $\beta_i(dh)$ by β_i. *cm* can be no more valid after this modification. Attributes of *dh* and $\beta_i(dh)$ are not modified.
Precondition: $0 \leq i \leq dimension$, $*dh \in cm.darts()$, and *dh* is not *i*-free.

advanced

advanced

Boolean marks

int *cm.get_new_mark() const*

Reserves a new mark. Returns its index. Returns -1 if there is no more available free mark.

bool	*cm.is_reserved(int amark) const*	
		Returns true iff *amark* is a reserved mark of *cm*. *Precondition*: 0≤*amark*<*NB_MARKS*.
bool	*cm.is_marked(Dart_const_handle dh, int amark) const*	
		Returns true iff *dh* is marked for *amark*. *Precondition*: *is_reserved(amark)* and **dh*∈*cm.darts()*.
void	*cm.mark(Dart_const_handle dh, int amark) const*	
		Marks *dh* for *amark*. *Precondition*: *is_reserved(amark)* and **dh*∈*cm.darts()*.
void	*cm.unmark(Dart_const_handle dh, int amark) const*	
		Unmarks *dh* for the mark *amark*. *Precondition*: *is_reserved(amark)* and **dh*∈*cm.darts()*.
void	*cm.negate_mark(int amark) const*	
		Inverse the mark *amark* for all the darts of *cm*. All the marked darts become unmarked and all the unmarked darts become marked. *Precondition*: *is_reserved(amark)*.
void	*cm.unmark_all(int amark) const*	
		Unmarks all the darts of *cm* for *amark*. *Precondition*: *is_reserved(amark)*.
size_type	*cm.number_of_marked_darts(int amark) const*	
		Returns the number of marked darts for *amark*. *Precondition*: *is_reserved(amark)*.
size_type	*cm.number_of_unmarked_darts(int amark) const*	
		Return the number of unmarked darts for *amark*. *Precondition*: *is_reserved(amark)*.
void	*cm.free_mark(int amark) const*	
		Frees *amark*. *Precondition*: *is_reserved(amark)*.

advanced

Has Models

CGAL::Combinatorial_map<d,Items,Alloc> . . . . page 1936

Concept

Dart

Definition

The concept Dart defines a d-dimensional dart. A dart mainly stores handles to the darts linked with itself by β_i, $\forall i$, $0 \leq i \leq d$. Moreover, it stores also handles to each non void attribute associated with itself.

Creation

A dart *d0* is never constructed directly, but always created within a combinatorial map *cm* by using the method *cm.create_dart();* A new dart is initialized to be i-free, $\forall i$: $0 \leq i \leq dimension$, and having all its attribute handles initialized to NULL, for each non void attribute.

Constants

static unsigned int *dimension;* The dimension of *d0*.

Types

Dart:: Dart_handle Dart handle type.
Dart:: Dart_const_handle Dart const handle type.

Dart:: template<unsigned int i> Attribute_handle<i>::type

Handle to i-attributes, with $0 \leq i \leq dimension$.

Dart:: template<unsigned int i> Attribute_const_handle<i>::type

Const handle to i-attributes, with $0 \leq i \leq dimension$.

Access Member Functions

Dart_handle *d0.beta(unsigned int i)* Returns $\beta_i(d0)$.
Precondition: $0 \leq i \leq dimension$.

Dart_const_handle *d0.beta(unsigned int i) const*

Returns $\beta_i(d0)$ when *d0* is const.
Precondition: $0 \leq i \leq dimension$.

Dart_handle *d0.beta_inv(unsigned int i)*

Returns $\beta_i^{-1}(d0)$.
Precondition: $0 \leq i \leq dimension$.

Dart_const_handle *d0.beta_inv(unsigned int i) const*

Returns $\beta_i^{-1}(d0)$ when *d0* is const.
Precondition: $0 \leq i \leq dimension$.

bool	*d0.is_free(unsigned int i) const*	
		Returns true iff *d0* is *i*-free. *Precondition*: $0 \leq i \leq dimension$.
int	*d0.highest_nonfree_dimension() const*	
		Returns the highest dimension *i* such that *d0* is not *i*-free. -1 if *d0* is free for any dimension.
Dart_handle	*d0.opposite()*	Returns a handle to a dart belonging to the same edge than *d0*, and not to the same vertex. NULL if such a dart does not exist.
Dart_const_handle	*d0.opposite() const*	Returns a handle to a dart belonging to the same edge than *d0*, and not to the same vertex, when *d0* is const. NULL if such a dart does not exist.
Dart_handle	*d0.other_extremity()*	Returns a handle to a dart belonging to the other vertex of the edge containing *d0* (but contrary to *opposite()* not necessarily to the same edge). NULL if such a dart does not exist.
Dart_const_handle	*d0.other_extremity() const*	
		Returns a *Dart_const_handle* to a dart belonging to the other vertex of the edge containing *d0*, when *d0* is const (but contrary to *opposite()* not necessarily to the same edge). NULL if such a dart does not exist.
template <unsigned int i> Attribute_handle<i>::type		
	d0.attribute()	Returns the *i*-attribute associated to *d0*. *Precondition*: $0 \leq i \leq dimension$, and *i*-attributes are non void.
template <unsigned int i> Attribute_const_handle<i>::type		
	d0.attribute() const	Returns the *i*-attribute associated to *d0*, when *d0* is const. *Precondition*: $0 \leq i \leq dimension$, and *i*-attributes are non void.

Has Models

CGAL::Dart<d,CMap> . . . . page 1938

CellAttribute

Definition

The concept CellAttribute represents a non void attribute associated with a cell of a combinatorial map. It can keep a handle to one dart of its associated cell, and can contain any information.

Types

CellAttribute:: Dart_handle	Dart handle type.
CellAttribute:: Dart_const_handle	Dart const handle type.
CellAttribute:: Info	Type of the information contained in the attribute. If *void*, the cell attribute does not have any information.
CellAttribute:: Supports_cell_dart	Equals to *Tag_true* to enable the storage of a *Dart_handle* of the associated cell, *Tag_false* otherwise.
CellAttribute:: On_merge	Functor called before merging two attributes. Must be a model of the *Binary Function* concept having two references to a model of *CellAttribute* as type of both arguments and *void* as return type.
CellAttribute:: On_split	Functor called after an attribute was split in two. Must be a model of the *Binary Function* concept having two references to a model of *CellAttribute* as type of both arguments and *void* as return type.

Creation

CellAttribute ca;

CellAttribute ca(Info info);	Constructor initializing the information of *ca* by the copy contructor *Info(info)*. Defined only if *Info* is different from *void*.

Access Member Functions

Dart_handle	*ca.dart()*	Returns one dart of the cell associated to attribute *ca*. NULL if *Supports_cell_dart* is equal to *Tag_false*.
Dart_const_handle	*ca.dart() const*	Returns one dart of the cell associated to attribute *ca*, when *ca* is const. NULL if *Supports_cell_dart* is equal to *Tag_false*.
void	*ca.set_dart(Dart_handle ahandle)*	
		Sets the dart of the cell associated to attribute *ca* to *ahandle*, if *Supports_cell_dart* is equal to *Tag_true*. Otherwise, this method does nothing. *Precondition*: *ahandle* belongs to the cell associated to *ca*.

Info&	*ca.info()*	Returns the information of *ca*. Defined only if *Info* is different from *void*.
Info	*ca.info() const*	Returns the information of *ca*, when *ca* is const. Defined only if *Info* is different from *void*.

Has Models

CGAL::Cell_attribute<CMap,Info_,Tag,OnMerge,OnSplit> . . . . page 1939

See Also

CombinatorialMap . . . . page 1919
CombinatorialMapItems . . . . page 1934

CombinatorialMapItems

Definition

The concept CombinatorialMapItems allows to customize a *d*D combinatorial map by choosing the type of darts, and by enabling and disabling some attributes. For that, it defines an inner class template named *Dart_wrapper*, with one template parameter, *CMap*, a model of the *CombinatorialMap* concept. This inner class must define two types: *Dart* and *Attributes*.

Types in *CombinatorialMapItems::Dart_wrapper<CMap>*

CombinatorialMapItems::Dart_wrapper<CMap>:: Dart

The type of dart, a model of the *Dart* concept.

CombinatorialMapItems::Dart_wrapper<CMap>:: Attributes

The tuple of attributes, containing at most *dimension+1* types (one for each possible cell of the combinatorial map). Each type of the tuple must be either a model of the *CellAttribute* concept or *void*. The first type corresponds to 0-attributes, the second to 1-attributes and so on. If the i^{th} type in the tuple is *void*, (i-1)-attributes are disabled. Otherwise, (i-1)-attributes are enabled and have the given type. If the size of the tuple is k, with *k*¡*dimension+1*, $\forall i$: $k \leq i \leq dimension$, i-attributes are disabled.

Example

The following examples show two possible models of the *CombinatorialMapItems* concept: the first one for a 4D combinatorial map without enabled attributes, the second one for a 3D combinatorial map with edge attributes enabled, and associated with a *Cell_attribute* containing an *int*.

```
struct Exemple_Item_4
{
   template < class CMap >
   struct Dart_wrapper
   {
     typedef CGAL::Dart<4, CMap> Dart;
     typedef CGAL::cpp0x::tuple<> Attributes;
   };
};
```

```
struct Exemple_Item_3
{
   template < class CMap >
   struct Dart_wrapper
   {
```

```
      typedef CGAL::Dart<3, CMap> Dart;
      typedef Cell_attribute<CMap, int> Edge_attrib;
      typedef CGAL::cpp0x::tuple<void,Edge_attrib> Attributes;
    };
};
```

Has Models

CGAL::Combinatorial_map_min_items<d> .... page 1940

See Also

CombinatorialMap .... page 1919
Dart .... page 1930

Class

CGAL::Combinatorial_map<d,Items,Alloc>

#include <CGAL/Combinatorial_map.h>

Definition

The class *Combinatorial_map<d,Items,Alloc>* represents a dD combinatorial map.

Darts and non void attributes are stored in memory using *CGAL::Compact_container*, using *Alloc* as allocator.

Is Model for the Concepts

CombinatorialMap . . . . page 1919

Parameters

d an integer for the dimension of the map.
Items must be a model of the *CombinatorialMapItems* concept.
Alloc has to match the standard allocator requirements. The *rebind* mechanism from *Alloc* will be used to create appropriate allocators internally with value type *Dart*.

There are two default template arguments: *Combinatorial_map_min_items<d>* for *Items* and *CGAL_ALLOCATOR(int)* from the *<CGAL/memory.h>* header file for *Alloc*.

Types

typedef Combinatorial_map<d,Items,Alloc>	*Self;*
typedef Items::Dart_wrapper<Self>::Dart	*Dart;*

Complexity

The complexity of *sew* and *unsew* is in $O(|S| \times |c|)$, S being the set of darts of the orbit $\langle \beta_1,\ldots,\beta_{i-2},\beta_{i+2},\ldots,\beta_d \rangle$ for the considered dart, and c the biggest j-cell merged or split during the sew such that j-attributes are non void. The complexity of *is_sewable* is in $O(|S|)$.

The complexity of *set_attribute* is in $O(|c|)$, c being the i-cell containing the considered dart.

The complexity of *is_without_boundary(unsigned int i)* is in $O(|D|)$, D being the set of darts of the combinatorial map, and the complexity of *is_without_boundary()* is in $O(|D| \times d)$.

The complexity of *unmark_all* and *free_mark* is in $O(1)$ if all the darts of the combinatorial map have the same mark, and in $O(|D|)$ otherwise.

The complexity of *is_valid* is in $O(|D| \times d^2)$.

The complexity of *clear* is in $O(|D| \times d)$.

Other methods have all a constant time complexity.

See Also

CombinatorialMapItems . . . . page 1934
Dart . . . . page 1930

Class

CGAL::Dart<d,CMap>

#include <CGAL/Dart.h>

Definition

The class *Dart<d,CMap>* represents a *d*D dart.

β_i pointers are coded in a array of *d+1 Dart_handle* (because we describe also the β_0 link). Attributes are associated to each dart by *Attribute_handle<i>*, one for each non void *i*-attribute.

Is Model for the Concepts

Dart . . . . page 1930

Parameters

d an integer for the dimension of the dart.
CMap must be a model of the *CombinatorialMap* concept.

Types

typedef CMap::Dart_handle	*Dart_handle;*
typedef CMap::Dart_const_handle	*Dart_const_handle;*
typedef CMap::Attribute_handle<i>::type	*Attribute_handle<i>::type;*
typedef CMap::Attribute_const_handle<i>::type	*Attribute_const_handle<i>::type;*

Complexity

Each β_i link is initialized to *CMap::null_dart_handle*, and each attribute handle of non void *i*-attribute is initialized to *NULL* at the creation of the dart, thus the complexity of the creation is in $O(d+1)$.

The complexity of *opposite* and *other_extremity* methods is in $O(d+1)$.

Other methods have all a constant time complexity.

See Also

CombinatorialMap . . . . page 1919

CGAL::Cell_attribute<CMap,Info_,Tag,OnMerge,OnSplit>

Class

#include <CGAL/Cell_attribute.h>

Definition

The class *Cell_attribute<CMap,Info_,Tag,OnMerge,OnSplit>* represents an attribute containing (or not) an information.

Is Model for the Concepts

CellAttribute . . . . page 1932

Parameters

CMap must be a model of the *CombinatorialMap*.
*Info_* is the type of the information contained in the attribute.
Tag is *Tag_true* to enable the storage of a *Dart_handle* of the associated cell, *Tag_false* otherwise.
OnMerge is the type of the functor called before two attributes are merged.
OnSplit is the type of the functor called after one attribute is split in two.

By default, *OnMerge* and *OnSplit* are equal to *Null_functor*; *Tag* is equal to *Tag_true*; and *Info_* is equal to *void*.

Types

*typedef Info_*	*Info;*
typedef Tag	*Supports_cell_dart;*
typedef OnMerge	*On_merge;*
typedef OnSplit	*On_split;*
typedef CMap::Dart_handle	*Dart_handle;*
typedef CMap::Dart_const_handle	*Dart_const_handle;*

See Also

CGAL::Combinatorial_map<d,Items,Alloc> . . . . page 1936

Class

CGAL::Combinatorial_map_min_items<d>

#include <CGAL/Combinatorial_map_min_items.h>

Definition

The class *Combinatorial_map_min_items<d>* is a model of the *CombinatorialMapItems* concept. It defines the type of darts which is a *CGAL::Dart<d,CMap>*. The *Combinatorial_map_min_items<d>* has a template argument for the dimension of the combinatorial map. In this class, no attribute is enabled.

Is Model for the Concepts

CombinatorialMapItems . . . . page 1934

Example

The following example shows the implementation of the *CGAL::Combinatorial_map_min_items* class.

```
template <unsigned int d>
struct Combinatorial_map_min_items
{
   template <class CMap>
   struct Dart_wrapper
   {
     typedef CGAL::Dart<d, CMap> Dart;
     typedef CGAL::cpp0x::tuple<> Attributes;
   };
};
```

See Also

CGAL::Combinatorial_map<d,Items,Alloc> . . . . page 1936
CGAL::Dart<d,CMap> . . . . page 1938

CGAL::make_edge<CMap>

#include <CGAL/Combinatorial_map_constructors.h>

template < class CMap >
typename CMap::Dart_handle

make_edge(CMap& cm)

Creates an isolated edge (two darts linked by β_2) and adds it in *cm*. Returns a handle on one dart of this edge.
Precondition: *CMap::dimension*$\geq$2.

See Also

CGAL::make_combinatorial_polygon<CMap> . . . . page 1942
CGAL::make_combinatorial_tetrahedron<CMap> . . . . page 1943
CGAL::make_combinatorial_hexahedron<CMap> . . . . page 1944

Function

CGAL::make_combinatorial_polygon<CMap>

#include <CGAL/Combinatorial_map_constructors.h>

template < class CMap >
typename CMap::Dart_handle

make_combinatorial_polygon(CMap& cm, unsigned int lg)

Creates a combinatorial polygon of length *lg* (*lg* darts linked by β_1), and adds it in *cm*. Returns a handle on one dart of this combinatorial polygon.
Precondition: *CMap::dimension*$\geq$1 and *lg*>0.

See Also

CGAL::make_edge<CMap> .... page 1941
CGAL::make_combinatorial_tetrahedron<CMap> .... page 1943
CGAL::make_combinatorial_hexahedron<CMap> .... page 1944

CGAL::make_combinatorial_tetrahedron<CMap>

#include <CGAL/Combinatorial_map_constructors.h>

template < class CMap >
typename CMap::Dart_handle

make_combinatorial_tetrahedron(CMap& cm)

Creates a combinatorial tetrahedron (four combinatorial triangles linked together by β_2), and adds it in *cm*. Returns a handle on one dart of this combinatorial tetrahedron.
Precondition: *CMap::dimension*$\geq$2.

Function

See Also

CGAL::make_edge<CMap> .... page 1941
CGAL::make_combinatorial_polygon<CMap> .... page 1942
CGAL::make_combinatorial_hexahedron<CMap> .... page 1944

Function

CGAL::make_combinatorial_hexahedron<CMap>

#include <CGAL/Combinatorial_map_constructors.h>

template < class CMap >
typename CMap::Dart_handle

make_combinatorial_hexahedron(CMap& cm)

Creates an combinatorial hexahedron (six combinatorial quadrangles linked together by β_2), and adds it in *cm*. Returns a handle on one dart of this combinatorial hexahedron.
Precondition: *CMap::dimension*$\geq$2.

See Also

CGAL::make_edge<CMap> .... page 1941
CGAL::make_combinatorial_polygon<CMap> .... page 1942
CGAL::make_combinatorial_tetrahedron<CMap> .... page 1943

CGAL::is_removable<CMap,i>

#include <CGAL/Combinatorial_map_operations.h>
template <class CMap, unsigned int i>

bool *is_removable(CMap cm, typename CMap::Dart_const_handle dh)*

Returns true iff the i-cell containing *dh* can be removed, i.e. if i==*CMap::dimension* or if i==*CMap::dimension*-1 or if i<*CMap::dimension*-1 and the i-cell containing *dh* is incident to at most two (i+*1*)-cells.
Precondition: $0 \leq i \leq$ *CMap::dimension* and **dh*∈*cm.darts()*.

See Also

CGAL::remove_cell<CMap,i> . . . . page 1946

Function

Function

CGAL::remove_cell<CMap,i>

#include <CGAL/Combinatorial_map_operations.h>

template <class CMap, unsigned int i>
unsigned int *remove_cell(CMap& cm, typename CMap::Dart_handle dh)*

Removes the *i*-cell containing *dh*. Returns the number of darts removed from *cm*.
Precondition: *is_removable<CMap,i>(cm,dh)*.
See examples in Figures 27.11, 27.13 and 27.14.
If *i<CMap::dimension*, and *i+1*-attributes are non void, and if there are two distinct (*i+1*)-cells around dart *dh*, *Attribute_type<i+1>::type::On_merge*(*a1*,*a2*) is called, with *a1* the (*i+1*)-attribute associated to *dh*, and *a2* the (*i+1*)-attribute associated to $\beta_{i+1}(dh)$.
If a *j*-cell is disconnected in two *j*-cells during the operation, and if *j*-attributes are non void, *Attribute_type<j>::type::On_split*(*a*,*a'*) is called with *a* the original *j*-attribute and *a'* the new *j*-attribute created due to the disconnection.

See Also

CGAL::is_removable<CMap,i> .... page 1945
CGAL::insert_cell_0_in_cell_1<CMap> .... page 1949
CGAL::insert_cell_0_in_cell_2<CMap> .... page 1950
CGAL::insert_cell_1_in_cell_2<CMap> .... page 1951
CGAL::insert_dangling_cell_1_in_cell_2<CMap> .... page 1952
CGAL::insert_cell_2_in_cell_3<CMap,InputIterator> .... page 1953

CGAL::is_insertable_cell_1_in_cell_2<CMap>

#include <CGAL/Combinatorial_map_operations.h>

template < class CMap >
bool *is_insertable_cell_1_in_cell_2(CMap cm,*
typename CMap::Dart_const_handle dh1,
typename CMap::Dart_const_handle dh2)

Returns true iff it is possible to insert a 1-cell in *cm* between *dh1* and *dh2*, i.e. if $dh1 \neq dh2$ and $dh1 \in \langle\beta_1\rangle(dh2)$.
Precondition: $CMap::dimension \geq 2$, $*dh1 \in cm.darts()$, and $*dh2 \in cm.darts()$.

Function

See Also

CGAL::insert_cell_1_in_cell_2<CMap> . . . . page 1951
CGAL::is_insertable_cell_2_in_cell_3<CMap,InputIterator> . . . . page 1948

Function

CGAL::is_insertable_cell_2_in_cell_3<CMap,InputIterator>

#include <CGAL/Combinatorial_map_operations.h>

template <class CMap, class InputIterator>
bool *is_insertable_cell_2_in_cell_3(CMap cm, InputIterator afirst, InputIterator alast)*

Returns true iff it is possible to insert a 2-cell in *cm* along the path of darts given by the range *[afirst,alast)*. The 2-cell can be inserted iff each couple of consecutive darts of the path *a1* and *a2* belong to the same vertex and the same volume, and if the path is closed.
Precondition: *CMap::dimension*$\geq$3.

See Also

CGAL::insert_cell_2_in_cell_3<CMap,InputIterator> . . . . page 1953
CGAL::is_insertable_cell_1_in_cell_2<CMap> . . . . page 1947

CGAL::insert_cell_0_in_cell_1<CMap>

#include <CGAL/Combinatorial_map_operations.h>

template < class CMap >
typename CMap::Dart_handle

insert_cell_0_in_cell_1(CMap& cm, typename CMap::Dart_handle dh)

Inserts a 0-cell in the 1-cell containing *dh*. Returns a handle on one dart belonging to the new 0-cell.
Precondition: *CMap::dimension*$\geq$1 and **dh*$\in$*cm.darts()*.
See example in Figure 27.11.
If 1-attributes are non void, *Attribute_type<1>::type::On_split*(*a*,*a'*) is called, with *a* the original 1-attribute associated with *dh* and *a'* the new 1-attribute created during the operation.

Function

See Also

CGAL::insert_cell_0_in_cell_2<CMap> . . . . page 1950
CGAL::insert_cell_1_in_cell_2<CMap> . . . . page 1951
CGAL::insert_dangling_cell_1_in_cell_2<CMap> . . . . page 1952
CGAL::insert_cell_2_in_cell_3<CMap,InputIterator> . . . . page 1953
CGAL::remove_cell<CMap,i> . . . . page 1946

Function

CGAL::insert_cell_0_in_cell_2<CMap>

#include <CGAL/Combinatorial_map_operations.h>

template <class CMap>
typename CMap::Dart_handle

insert_cell_0_in_cell_2(CMap & cm, typename CMap::Dart_handle dh)

Inserts a 0-cell in the 2-cell containing *dh*. The 2-cell is split in triangles, one for each initial edge of the facet. Returns a handle on one dart belonging to the new 0-cell.
Precondition: *CMap::dimension*$\geq$2 and **dh*$\in$*cm.darts()*.
See example in Figure 27.12.
If 2-attributes are non void, *Attribute_type<2>::type::On_split*(*a*,*a'*) is called, with *a* the original 2-attribute associated with *dh* and *a'* each new 2-attribute created during the operation.

See Also

CGAL::insert_cell_0_in_cell_2<CMap> . . . . page 1950
CGAL::insert_cell_1_in_cell_2<CMap> . . . . page 1951
CGAL::insert_dangling_cell_1_in_cell_2<CMap> . . . . page 1952
CGAL::insert_cell_2_in_cell_3<CMap,InputIterator> . . . . page 1953
CGAL::remove_cell<CMap,i> . . . . page 1946

CGAL::insert_cell_1_in_cell_2<CMap>

#include <CGAL/Combinatorial_map_operations.h>

template < class CMap >
typename CMap::Dart_handle

insert_cell_1_in_cell_2(CMap& cm,
typename CMap::Dart_handle dh1,
typename CMap::Dart_handle dh2)

Inserts a 1-cell in the 2-cell containing *dh1* and *dh2*. Returns β_0(*dh1*), a handle on one dart belonging to the new 1-cell.
Precondition: *is_insertable_cell_1_in_cell_2<Map>(cm,dh1,dh2)*.
See example in Figure 27.13.
If 2-attributes are non void, *Attribute_type<2>::type::On_split*(*a*,*a'*) is called, with *a* the original 2-attribute associated with *dh* and *a'* the new 2-attribute created during the operation.

See Also

Function

CGAL::insert_dangling_cell_1_in_cell_2<CMap>

#include <CGAL/Combinatorial_map_operations.h>

template < class CMap >
typename CMap::Dart_handle

insert_dangling_cell_1_in_cell_2(CMap& cm, typename CMap::Dart_handle dh)

Inserts a 1-cell in a the 2-cell containing *dh*, the 1-cell being attached only by one of its extremity to the 0-cell containing *dh*. Returns a handle on one dart belonging to the new 1-cell.
Precondition: *CMap::dimension*$\geq$2 and **dh*$\in$*cm.darts()*.
See example in Figure 27.13.

See Also

CGAL::insert_cell_2_in_cell_3<CMap,InputIterator>

Function

#include <CGAL/Combinatorial_map_operations.h>

template <class CMap, class InputIterator>
typename CMap::Dart_handle

insert_cell_2_in_cell_3(CMap & cm, InputIterator afirst, InputIterator alast)

Inserts a 2-cell along the path of 1-cells containing darts given by the range *[afirst,alast)*. Returns a handle on one dart belonging to the new 2-cell.
Precondition: *is_insertable_cell_2_in_cell_3<Map>(cm,afirst,alast)*.
See example in Figure 27.14.
If 3-attributes are non void, *Attribute_type<3>::type::On_split*(*a*,*a'*) is called, with *a* the original 3-attribute associated with *dh* and *a'* the new 3-attribute created during the operation.

See Also

CGAL::is_insertable_cell_2_in_cell_3<CMap,InputIterator> . . . . page 1948
CGAL::insert_cell_0_in_cell_1<CMap> . . . . page 1949
CGAL::insert_cell_0_in_cell_2<CMap> . . . . page 1950
CGAL::insert_cell_1_in_cell_2<CMap> . . . . page 1951
CGAL::insert_dangling_cell_1_in_cell_2<CMap> . . . . page 1952
CGAL::remove_cell<CMap,i> . . . . page 1946

Chapter 28

Linear Cell Complex

Guillaume Damiand

Contents

28.1 Introduction

A *d*D linear cell complex allows to represent an orientable subdivided *d*D object having linear geometry: each vertex of the subdivision is associated with a point. The geometry of each edge is a segment whose end points are associated with the two vertices of the edge, the geometry of each 2-cell is obtained from all the segments associated to the edges describing the boundary of the 2-cell and so on.

The combinatorial part of a linear cell complex is described by a *d*D combinatorial map (it is strongly recommended to first read the combinatorial maps chapter (Page 1887) for definitions). To add the linear geometrical embedding, a point (a model of *CGAL::Point_2* or *CGAL::Point_3* or *CGAL::Point_d*) is associated to each vertex of the combinatorial map.

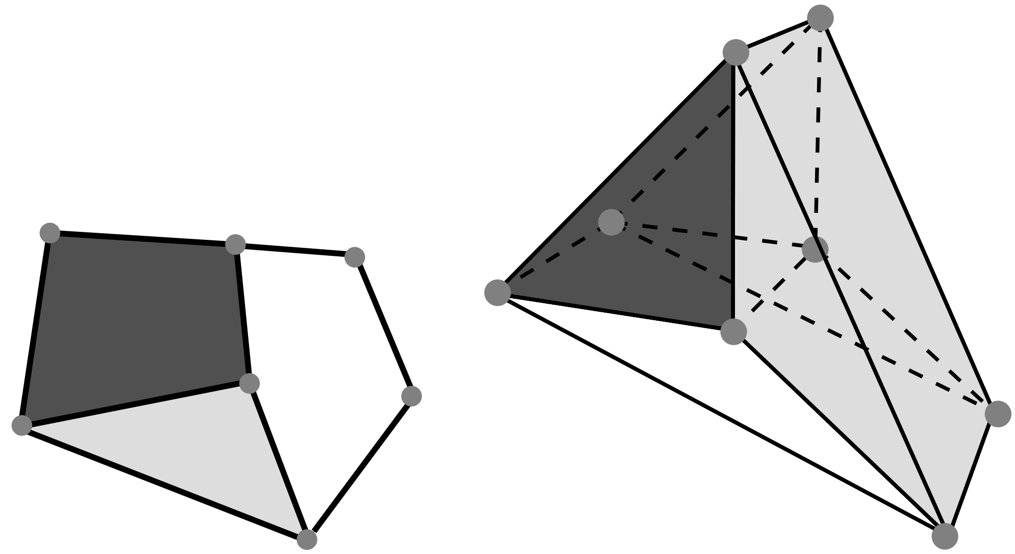


Figure 28.1: Examples of objects with linear geometry. **Left**: A 2D object composed of three 2-cells, nine 1-cells and seven points associated to the seven 0-cells . **Right**: A 3D object composed of three 3-cells, twelve 2-cells, sixteen 1-cells and eight points associated to the eight 0-cells.

If we reconsider the example introduced in the combinatorial map package, recalled in Figure 28.1 (Right), the combinatorial part of the 3D object is described by a 3D combinatorial map. As illustrated in Figure 28.2, the geometrical part of the object is described by associating a point to each vertex of the map.

Note that the dimension of the combinatorial map *d* is not necessarily equal to the dimension of the ambient space *d2*. Indeed, we can use for example a 2D combinatorial map in a 2D ambient space to describe a planar graph (*d*=*d2*=*2*), or a 2D combinatorial map in a 3D ambient space to describe a surface in 3D space (*d*=2, *d2*=3) (case of the *Polyhedron_3* package), or a 3D combinatorial map in a 3D ambient space (*d*=*d2*=3) and so on.

28.2 Software Design

The diagram in Figure 28.3 shows the main classes of the package. *CGAL::Linear_cell_complex* is the main class (see Section 28.2.1), which inherits from the *CGAL::Combinatorial_map* class. Attributes can be associated to some cells of the linear cell complex thanks to an items class (see Section 28.2.4), which defines the dart type and the attributes types. These types may be different for different dimensions of cells, and they may also be void. In the class *CGAL::Linear_cell_complex*, it is required that specific attributes are associated to all vertices of the combinatorial map. These attributes must contain a point (a model of *CGAL::Point_2* or *CGAL::Point_3* or *CGAL::Point_d*), and can be represented by instances of class *CGAL::Cell_attribute_with_point* (see Section 28.2.2).

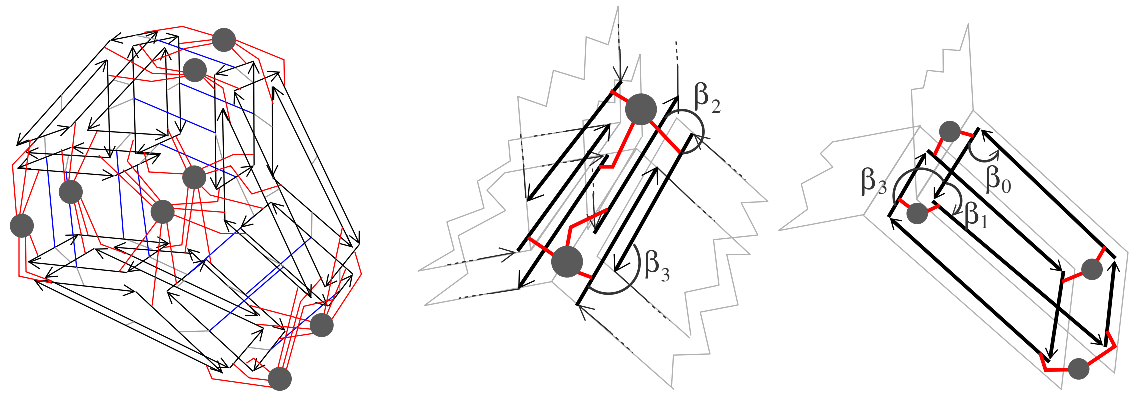


Figure 28.2: Example of 3D linear cell complex describing the object given in Figure 28.1 (Right). **Left**: The 3D linear cell complex which contains 54 darts (18 for each 3-cell) where each vertex is associated with a point, here a *CGAL::Point_3*. Blue segments represent β_3 relations. **Middle**: Zoom around the central edge which details the six darts belonging to the edge and the associations between darts and points. **Right**: Zoom around the facet between light gray and white 3-cells, which details the eight darts belonging to the facet and the associations between darts and points (given by red segments).

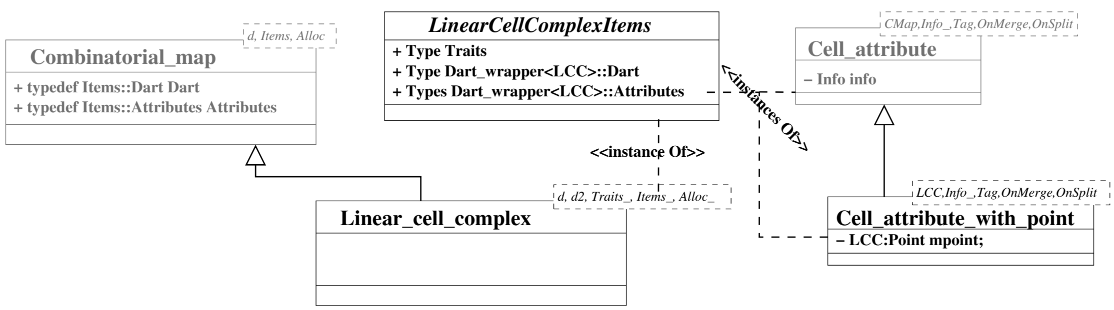


Figure 28.3: UML diagram of the main classes of the package. Gray elements come from the combinatorial map package (Page 1887).

28.2.1 Linear Cell Complex

The *CGAL::Linear_cell_complex<d,d2,LCCTraits,Items,Alloc>* class is a model of the *CombinatorialMap* concept. It guarantees that each vertex of the combinatorial map is associated with an attribute containing a point. This class can be used in geometric algorithms (it plays the same role as *Polyhedron_3* for *HalfedgeDS*).

This class has five template parameters standing for the dimension of the combinatorial map, the dimension of the ambient space, a traits class (a model of the *LinearCellComplexTraits* concept, see Section 28.2.3), an items class (a model of the *LinearCellComplexItems* concept, see Section 28.2.4), and an allocator which must be a model of the allocator concept of STL. Default classes are provided for the traits, items, and for the allocator classes, and by default *d2=d*.

A linear cell complex is valid, if it is a valid combinatorial map where each dart is associated with an attribute containing a point (i.e. an instance of a model of the *CellAttributeWithPoint* concept). Note that there are no validity constraints on the geometry (test on self intersection, planarity of 2-cells...). We can see two examples of *CGAL::Linear_cell_complex* in Figure 28.4.

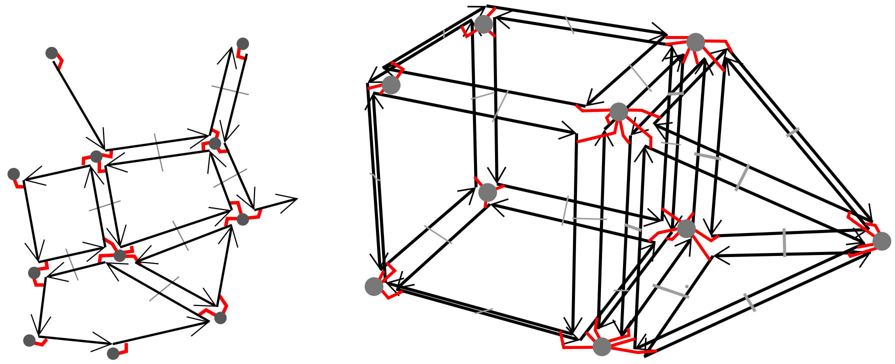


Figure 28.4: Examples of *CGAL::Linear_cell_complex*. Gray disks show the attributes associated to vertices. Associations between darts and attributes are drawn by small lines between darts and disks. **Left:** Example of *CGAL::Linear_cell_complex<2,2>*. **Right:** Example of *CGAL::Linear_cell_complex<3,3>*.

28.2.2 Cell Attributes

The *CGAL::Cell_attribute_with_point<LCC,Info_,Tag,OnMerge,OnSplit>* class is a model of the *CellAttributeWithPoint* concept, which is a refinement of the *CellAttribute* concept. It represents an attribute associated with a cell, which can contain an information (depending on whether *Info_==void* or not), but which always contains a point, an instance of *LCC::Point*.

28.2.3 Linear Cell Complex Traits

The *LinearCellComplexTraits* geometric traits concept defines the required types and functors used in the *Linear_cell_complex* class. For example it defines *Point*, the type of points used, and *Vector*, the corresponding vector type. It also defines all the required functors used for constructions and operations, as for example *Construct_translated_point* or *Construct_sum_of_vectors*.

The class *CGAL::Linear_cell_complex_traits<d,K>* is a model of *LinearCellComplexTraits*. It defines the different types which are obtained from *K* that, depending on *d*, is a model of the concept *Kernel* if *d==2* or *d==3*, and a model of the concept *Kernel_d* otherwise.

28.2.4 Linear Cell Complex Items

The *LinearCellComplexItems* concept refines the *CombinatorialMapItems* concept by adding the requirement that 0-attributes are enabled, and associated with a type of attribute being a model of the *CellAttributeWithPoint* concept.

The class *CGAL::Linear_cell_complex_min_items<d>* is a model of *LinearCellComplexItems*. It uses *CGAL::Dart<d>*, and instances of *CGAL::Cell_attribute_with_point* (which contain no information) associated to each vertex. All other attributes are void.

28.3 Operations

Several operations defined in the combinatorial maps package can be used on a linear cell complex. This is the case for all the iteration operations that do not modify the model (see example in Section 28.4.1). This is also

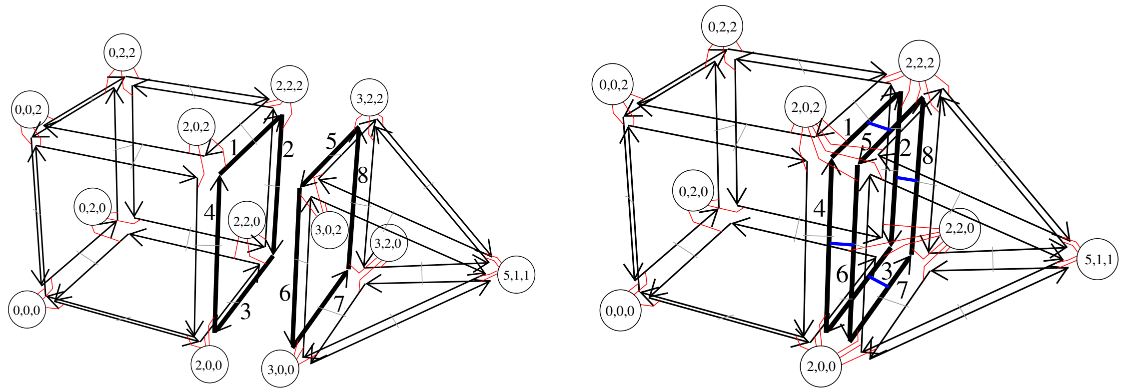


Figure 28.5: Example of 3-sew operation for linear cell complex. **Left**: A 3D linear cell complex containing two 3-cells that are not connected. Vertex attributes are drawn with circles containing point coordinates. Associations between darts and attributes are drawn with small lines between darts and disks. **Right**: The 3D linear cell complex obtained as result of *sew<3>(1,5)* (or *sew<3>(2,8)*, or *sew<3>(3,7)*, or *sew<3>(4,6)*). The eight 0-attributes around the facet between the two 3-cells before the sew operation, are merged into four 0-attributes after. The geometry of the pyramid is deformed since its base is fitted on the 2-cell of the cube.

the case for all the operations that do not create new 0-cells: *sew*, *unsew*, *remove_cell*, *insert_cell_1_in_cell_2*, *insert_cell_2_in_cell_3*. Indeed, all these operations update non void attributes, and thus update vertex attributes of a linear cell complex. Note that some existing 0-attributes can be duplicated by the *unsew* method, but these 0-attributes are not new but copies of existing old 0-attributes.

However, operations that create a new 0-cell can not be directly used since the new 0-cell would not be associated with a vertex attribute. Indeed, it is not possible for these operations to automatically decide which point to create. These operations are: *insert_cell_0_in_cell_1*, *insert_cell_0_in_cell_2 insert_dangling_cell_1_in_cell_2*, plus all the creation operations. For these operations, new versions are proposed taking some points as additional parameters. Lastly, some new operations are defined, which use the geometry (see sections 28.3.2 and 28.3.3).

All the operations given in this section guarantee that given a valid linear cell complex and a possible operation, the result is a valid linear cell complex. As for a combinatorial map, it is also possible to use low level operations but additional operations may be needed to restore the validity conditions.

28.3.1 Sewing and Unsewing

As explained in the combinatorial map user manual, Section 27.5.1, it is possible to glue two *i*-cells along an (*i*-1)-cell by using the *sew<i>* method. Since this method updates non void attributes, and since points are specific attributes, they are automatically updated during the *sew<i>* method. Thus the sewing of two *i*-cells could deform the geometry of the concerned objects.

For example, in Figure 28.5, we want to 3-sew the two initial 3-cells. *sew<3>(1,5)* links by β_3 the pairs of darts (1,5), (2,8), (3,7) and (4,6). The eight vertex attributes around the facet between the two 3-cells before the sew are merged by pair during the sew operation (and the *On_merge* functor is called four times). Thus, after the sew, there are only four 0-attributes around the facet. By default, the attributes associated with the first dart of the sew operation are kept (but this can be modified by defining your own functor in the attribute class as explained in the package combinatorial map, Section 27.5.1). Intuitively, the geometry of the second 2-cell is deformed to fit to the first 2-cell.

This is similar for the unsew operation, which removes β_i links of all the darts in $\langle\beta_1,\ldots,\beta_{i-2},\beta_{i+2},\ldots,\beta_d\rangle(d0)$, and updates non void attributes which are no more associated to a same cell due to the unlinks. If we take the linear cell complex given in Figure 28.5 (Right), and we call *unsew<3>(2)*, we obtain the linear cell complex in Figure 28.5 (Left) except for the coordinates of the new four vertices, which by default are copies of original vertices (this behavior can be modified thanks to the functor *On_split* in the attribute class). The *unsew<3>* operation has removed the four β_3 links, and has duplicated the 0-attributes since vertices are split in two after the unsew operation.

28.3.2 Construction Operations

There are several member functions allowing to insert specific configurations of darts into a linear cell complex. These functions return a *Dart_handle* to the new object. Note that the dimension of the linear cell complex must be large enough: darts must contain all the β used by the operation. All these methods add new darts in the current linear cell complex, existing darts are not modified. These functions are *make_segment*, *make_triangle*, *make_tetrahedron* and *make_hexahedron*.

There are two functions allowing to build a linear cell complex from two other CGAL data types:

- *import_from_triangulation_3(lcc,atr)*: adds in *lcc* all the tetrahedra present in *atr*, a *CGAL::Triangulation_3*;
- *import_from_polyhedron_3(lcc,ap)*: adds in *lcc* all the cells present in *ap*, a *CGAL::Polyhedron_3*.

Lastly, the function *import_from_plane_graph(lcc,ais)* adds in *lcc* all the cells reconstructed from the planar graph read in *ais*, a *std::istream* (see the reference manual for the file format).

28.3.3 Modification Operations

Some methods are defined in *Linear_cell_complex* class to modify a linear cell complex and update the vertex attributes. In the following, we denote by *dh0*, *dh1*, *dh2* the dart handles for the darts *d0*, *d1*, *d2*, respectively. That is *d0 == *dh0*.

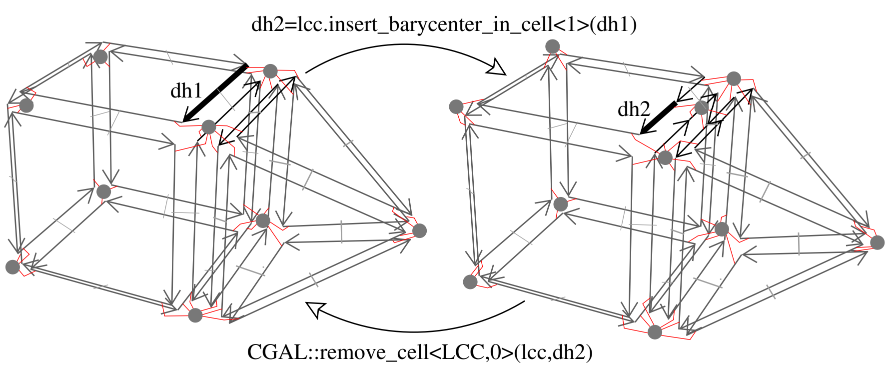


Figure 28.6: Example of *insert_barycenter_in_cell<1>* and *remove_cell<0>* operations. **Left**: Initial linear cell complex. **Right**: After the insertion of a point in the barycenter of the 1-cell containing dart *d1*. Now if we remove the 0-cell containing dart *d2*, we obtain a linear cell complex isomorphic to the initial one.

lcc.insert_barycenter_in_cell<unsigned int i>(dh0) adds the barycenter of the *i*-cell containing dart *d0*. This operation is possible if *d0∈lcc.darts()* (see examples on Figure 28.6 and Figure 28.7).

lcc.insert_point_in_cell<unsigned int i>(dh0,p) is an operation similar to the previous operation, the only difference being that the coordinates of the new point are here given by *p* instead of being computed as the barycenter of the *i*-cell. Currently, these two operations are only defined for *i=1* to insert a point in an edge, or *i=2* to insert a point in a facet.

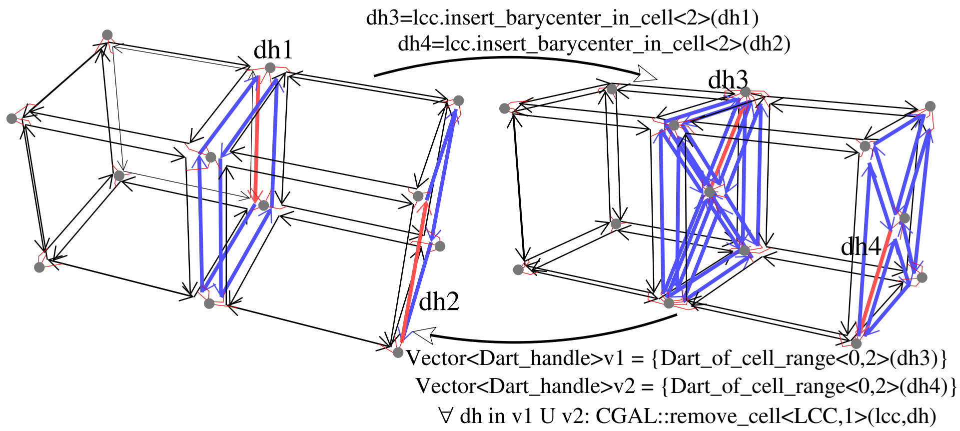


Figure 28.7: Examples of *insert_barycenter_in_cell<2>* operation.

lcc.insert_dangling_cell_1_in_cell_2(dh0,p) adds a 1-cell in the 2-cell containing dart *d0*, the 1-cell being attached by only one of its vertex to the 0-cell containing dart *d0*. The second vertex of the new edge is associated with a new 0-attribute containing a copy of *p* as point. This operation is possible if *d0∈lcc.darts()* (see example on Figure 28.8).

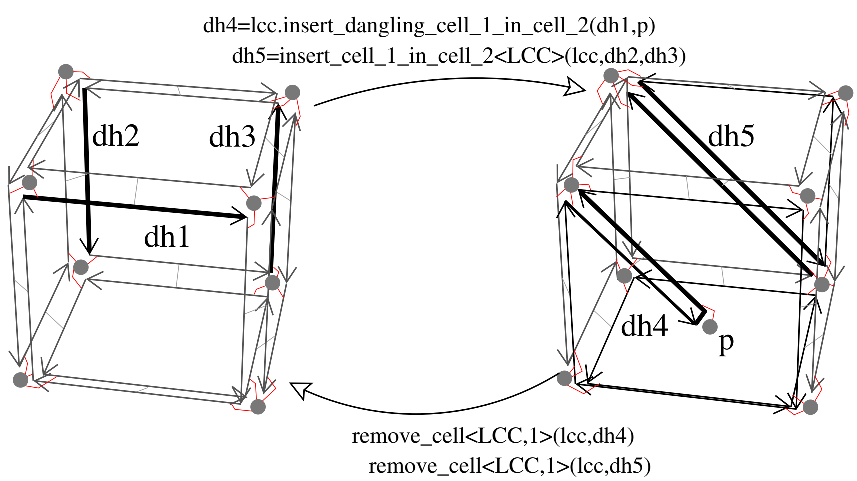


Figure 28.8: Example of *insert_dangling_cell_1_in_cell_2*, *insert_cell_1_in_cell_2* and *remove_cell<1>* operations. **Left**: Initial linear cell complex. **Right**: After the insertion of a dangling 1-cell in the 2-cell containing dart *d1*, and of a 1-cell in the 2-cell containing dart *d2*. Now if we remove the 1-cells containing dart *d4* and *d5*, we obtain a linear cell complex isomorphic to the initial one.

Some examples of use of these operations are given in Section 28.4.2.

28.4 Examples

28.4.1 A 3D Linear Cell Complex

This example uses a 3-dimensional linear cell complex. It creates two tetrahedra and displays all the points of the linear cell complex thanks to a *Vertex_attribute_const_range*. Then, the two tetrahedra are 3-sewn and we translate all the points of the second tetrahedron along vector *v(3,1,1)*. Since the two tetrahedra are 3-sewn, this translation moves also the 2-cell of the first tetrahedron shared with the second one. This is illustrated by displaying all the points of each 3-cell. For that we use a *std::for_each* and the *Display_vol_vertices* functor.

```
#include <CGAL/Linear_cell_complex.h>
#include <CGAL/Linear_cell_complex_operations.h>
#include <iostream>
#include <algorithm>

typedef CGAL::Linear_cell_complex<3> LCC_3;
typedef LCC_3::Dart_handle           Dart_handle;
typedef LCC_3::Point                 Point;
typedef LCC_3::FT                    FT;

// Functor used to display all the vertices of a given volume.
template<class LCC>
struct Display_vol_vertices : public std::unary_function<LCC, void>
{
  Display_vol_vertices(const LCC& alcc) :
    lcc(alcc),
    nb_volume(0)
  {}

  void operator() (typename LCC::Dart& d)
  {
    std::cout<<"Volume "<<++nb_volume<<" : ";
    for (typename LCC::template One_dart_per_incident_cell_range<0,3>::
           const_iterator it=lcc.template one_dart_per_incident_cell<0,3>
           (lcc.dart_handle(d)).begin(),
           itend=lcc.template one_dart_per_incident_cell<0,3>
           (lcc.dart_handle(d)).end();
         it!=itend; ++it)
    {
      std::cout << LCC_3::point(it) << "; ";
    }
    std::cout<<std::endl;
  }
private:
  const LCC& lcc;
  unsigned int nb_volume;
};

int main()
{
  LCC_3 lcc;

  // Create two tetrahedra.
```

```
  Dart_handle d1 = lcc.make_tetrahedron(Point(-1, 0, 0), Point(0, 2, 0),
                                        Point(1, 0, 0), Point(1, 1, 2));
  Dart_handle d2 = lcc.make_tetrahedron(Point(0, 2, -1),
                                        Point(-1, 0, -1),
                                        Point(1, 0, -1),
                                        Point(1, 1, -3));

  // Display all the vertices of the lcc by iterating on the
  // Vertex_attribute container.
  CGAL::set_ascii_mode(std::cout);
  std::cout<<"Vertices: ";
  for (LCC_3::Vertex_attribute_const_range::iterator
         v=lcc.vertex_attributes().begin(),
         vend=lcc.vertex_attributes().end();
       v!=vend; ++v)
    std::cout << v->point() << "; ";
  std::cout<<std::endl;

  // Display the vertices of each volume by iterating on darts.
  std::for_each(lcc.one_dart_per_cell<3>().begin(),
                lcc.one_dart_per_cell<3>().end(),
                Display_vol_vertices<LCC_3>(lcc));

  // 3-Sew the 2 tetrahedra along one facet
  lcc.sew<3>(d1, d2);

  // Display the vertices of each volume by iterating on darts.
  std::for_each(lcc.one_dart_per_cell<3>().begin(),
                lcc.one_dart_per_cell<3>().end(),
                Display_vol_vertices<LCC_3>(lcc));

  // Translate the second tetrahedra by a given vector
  LCC_3::Vector v(3,1,1);
  for (LCC_3::One_dart_per_incident_cell_range<0,3>::iterator
         it=lcc.one_dart_per_incident_cell<0,3>(d2).begin(),
         itend=lcc.one_dart_per_incident_cell<0,3>(d2).end();
       it!=itend; ++it)
  {
    LCC_3::point(it)=LCC_3::Traits::Construct_translated_point_3()
      (LCC_3::point(it),v);
  }

  // Display the vertices of each volume by iterating on darts.
  std::for_each(lcc.one_dart_per_cell<3>().begin(),
                lcc.one_dart_per_cell<3>().end(),
                Display_vol_vertices<LCC_3>(lcc));

  // We display the lcc characteristics.
  std::cout<<"LCC characteristics: ";
  lcc.display_characteristics(std::cout) << ", valid=" << lcc.is_valid()
                                         << std::endl;

  return EXIT_SUCCESS;
}
```

File: examples/Linear_cell_complex/linear_cell_complex_3.cpp

The output is:

```
Vertices: 1 1 2; 1 0 0; 0 2 0; -1 0 0; 1 1 -3; 1 0 -1; -1 0 -1; 0 2 -1;
Volume 1 : -1 0 0; 0 2 0; 1 0 0; 1 1 2;
Volume 2 : 0 2 -1; -1 0 -1; 1 0 -1; 1 1 -3;
Volume 1 : -1 0 0; 0 2 0; 1 0 0; 1 1 2;
Volume 2 : 0 2 0; -1 0 0; 1 0 0; 1 1 -3;
Volume 1 : 2 1 1; 3 3 1; 4 1 1; 1 1 2;
Volume 2 : 3 3 1; 2 1 1; 4 1 1; 4 2 -2;
LCC characteristics: #Darts=24, #0-cells=5, #1-cells=9, #2-cells=7, #3-cells=2, #ccs=1, valid=1
```

The first line gives the points of the linear cell complex before the *sew<3>*. There are eight points, four for each tetrahedron. After the sew, six vertices are merged two by two, thus there are five vertices. We can see the points of each 3-cell (lines Volume 1 and Volume 2) before the sew, after the sew and after the translation of the second volume. We can see that this translation has also modified the three common points between the two 3-cells. The last line shows the number of cells of the linear cell complex, the number of connected components, and finally a Boolean to show the validity of the linear cell complex.

28.4.2 A 4D Linear Cell Complex

This example uses a 4-dimensional linear cell complex embedded in a 5-dimensional ambient space. It creates two tetrahedra having 5D points and sews the two tetrahedra by β_4. Then we use some high level operations, display the number of cells of the linear cell complex, and check its validity. Last we use the reverse operations to get back to the initial configuration.

```
#include <CGAL/Linear_cell_complex.h>
#include <CGAL/Linear_cell_complex_constructors.h>
#include <iostream>
#include <vector>

typedef CGAL::Linear_cell_complex<4,5> LCC_4;
typedef LCC_4::Dart_handle                Dart_handle;
typedef LCC_4::Point                      Point;
typedef LCC_4::Vector                     Vector;
typedef LCC_4::FT                         FT;

int main()
{
  LCC_4 lcc;

  // Create two tetrahedra.
  FT p1[5]={ 0, 0, 0, 0, 0}; std::vector<FT> v1(p1,p1+5);
  FT p2[5]={ 0, 2, 0, 0, 0}; std::vector<FT> v2(p2,p2+5);
  FT p3[5]={ 0, 1, 2, 0, 0}; std::vector<FT> v3(p3,p3+5);
  FT p4[5]={ 2, 1, 0, 0, 0}; std::vector<FT> v4(p4,p4+5);
  FT p5[5]={-1, 0, 0, 0, 0}; std::vector<FT> v5(p5,p5+5);
  FT p6[5]={-1, 2, 0, 0, 0}; std::vector<FT> v6(p6,p6+5);
  FT p7[5]={-1, 1, 2, 0, 0}; std::vector<FT> v7(p7,p7+5);
  FT p8[5]={-3, 1, 2, 0, 0}; std::vector<FT> v8(p8,p8+5);
```

```
  Dart_handle d1 = lcc.make_tetrahedron(Point(5, v1.begin(), v1.end()),
                                       Point(5, v2.begin(), v2.end()),
                                       Point(5, v3.begin(), v3.end()),
                                       Point(5, v4.begin(), v4.end()));

  Dart_handle d2 = lcc.make_tetrahedron(Point(5, v5.begin(), v5.end()),
                                       Point(5, v6.begin(), v6.end()),
                                       Point(5, v7.begin(), v7.end()),
                                       Point(5, v8.begin(), v8.end()));

  lcc.display_characteristics(std::cout);
  std::cout<<", valid="<<lcc.is_valid()<<std::endl;

  lcc.sew<4>(d1,d2);

  lcc.display_characteristics(std::cout);
  std::cout<<", valid="<<lcc.is_valid()<<std::endl;

  // Add one vertex on the middle of the edge containing dart d1.
  Dart_handle d3 = lcc.insert_barycenter_in_cell<1>(d1);
  CGAL_assertion( lcc.is_valid() );

  lcc.display_characteristics(std::cout);
  std::cout<<", valid="<<lcc.is_valid()<<std::endl;

  // Add one edge to cut the face containing dart d3 in two.
  Dart_handle d4 = CGAL::insert_cell_1_in_cell_2(lcc,d3,d1->beta(0));
  CGAL_assertion( lcc.is_valid() );

  lcc.display_characteristics(std::cout);
  std::cout<<", valid="<<lcc.is_valid()<<std::endl;

  // We use removal operations to get back to the initial configuration.
  CGAL::remove_cell<LCC_4,1>(lcc,d4);
  CGAL_assertion( lcc.is_valid() );

  CGAL::remove_cell<LCC_4,0>(lcc,d3);
  CGAL_assertion( lcc.is_valid() );

  lcc.unsew<4>(d1);

  lcc.display_characteristics(std::cout);
  std::cout<<", valid="<<lcc.is_valid()<<std::endl;

  return EXIT_SUCCESS;
}
```

File: examples/Linear_cell_complex/linear_cell_complex_4.cpp

The output is:

```
#Darts=24, #0-cells=8, #1-cells=12, #2-cells=8, #3-cells=2, #4-cells=2, #ccs=2, valid=1
```

```
#Darts=24, #0-cells=4, #1-cells=6, #2-cells=4, #3-cells=1, #4-cells=2, #ccs=1, valid=1
#Darts=28, #0-cells=5, #1-cells=7, #2-cells=4, #3-cells=1, #4-cells=2, #ccs=1, valid=1
#Darts=32, #0-cells=5, #1-cells=8, #2-cells=5, #3-cells=1, #4-cells=2, #ccs=1, valid=1
#Darts=24, #0-cells=8, #1-cells=12, #2-cells=8, #3-cells=2, #4-cells=2, #ccs=2, valid=1
```

28.4.3 A 3D Linear Cell Complex with Colored Vertices

This example illustrates the way to use a 3D linear cell complex by adding another information to vertices. For that, we need to define our own items class. The difference with the *CGAL::Linear_cell_complex_min_items* class is about the definition of the vertex attribute where we use a *CGAL::Cell_attribute_with_point* with a non void info. In this example, the "vextex color" is just given by an *int* (the second template parameter of the *CGAL::Cell_attribute_with_point*). Lastly, we define the *Average_functor* class in order to set the color of a vertex resulting of the merging of two vertices to the average of the two initial values. This functor is associated with the vertex attribute by passing it as template parameter. Using this items class instead of the default one is done during the instantiation of template parameters of the *CGAL::Linear_cell_complex* class.

Now we can use *LCC_3* in which each vertex is associated with an attribute containing both a point and an information. In the following example, we create two cubes, and set the color of the vertices of the first cube to 1 and of the second cube to 19 (by iterating through two *One_dart_per_incident_cell_range<0, 3>* ranges). Then we *3-sew* the two cubes along one facet. This operation merges some vertices (as in the example of Figure 28.5). We insert a vertex in the common 2-cell between the two cubes, and set the information of the new 0-attribute to 5. In the last loop, we display the point and the information of each vertex of the linear cell complex.

```
#include <CGAL/Linear_cell_complex.h>
#include <CGAL/Linear_cell_complex_operations.h>
#include <iostream>
#include <algorithm>

struct Average_functor
{
  template<class CellAttribute>
  void operator()(CellAttribute& ca1,const CellAttribute& ca2)
  { ca1.info()=(ca1.info()+ ca2.info())/2; }
};

struct Myitem
{
  template<class Refs>
  struct Dart_wrapper
  {
    typedef CGAL::Dart<3, Refs > Dart;

    typedef CGAL::Cell_attribute_with_point< Refs, int, CGAL::Tag_true,
                                             Average_functor >
    Vertex_attribute;

    typedef CGAL::cpp0x::tuple<Vertex_attribute> Attributes;
  };
};

typedef CGAL::Linear_cell_complex_traits
<3, CGAL::Exact_predicates_inexact_constructions_kernel> Traits;
typedef CGAL::Linear_cell_complex<3,3,Traits,Myitem>     LCC_3;
```

```
typedef LCC_3::Dart_handle                                    Dart_handle;
typedef LCC_3::Point                                          Point;
typedef LCC_3::FT                                             FT;

Dart_handle make_iso_cuboid(LCC_3& lcc, const Point& basepoint, FT lg)
{
  return lcc.make_hexahedron(basepoint,
                             Traits::Construct_translated_point()
                             (basepoint,Traits::Vector(lg,0,0)),
                             Traits::Construct_translated_point()
                             (basepoint,Traits::Vector(lg,lg,0)),
                             Traits::Construct_translated_point()
                             (basepoint,Traits::Vector(0,lg,0)),
                             Traits::Construct_translated_point()
                             (basepoint,Traits::Vector(0,lg,lg)),
                             Traits::Construct_translated_point()
                             (basepoint,Traits::Vector(0,0,lg)),
                             Traits::Construct_translated_point()
                             (basepoint,Traits::Vector(lg,0,lg)),
                             Traits::Construct_translated_point()
                             (basepoint,Traits::Vector(lg,lg,lg)));
}

int main()
{
  LCC_3 lcc;

  // Create two iso_cuboids.
  Dart_handle d1 = make_iso_cuboid(lcc, Point(-2, 0, 0), 1);
  Dart_handle d2 = make_iso_cuboid(lcc, Point(0, 0, 0), 1);

  // Set the "color" of all vertices of the first cube to 1.
  for (LCC_3::One_dart_per_incident_cell_range<0, 3>::iterator
         it=lcc.one_dart_per_incident_cell<0,3>(d1).begin(),
         itend=lcc.one_dart_per_incident_cell<0,3>(d1).end(); it!=itend;
   ++it)
  { LCC_3::vertex_attribute(it)->info()=1; }

  // Set the "color" of all vertices of the second cube to 19.
  for (LCC_3::One_dart_per_incident_cell_range<0, 3>::iterator it=
         lcc.one_dart_per_incident_cell<0,3>(d2).begin(),
         itend=lcc.one_dart_per_incident_cell<0,3>(d2).end(); it!=itend;
   ++it)
  { LCC_3::vertex_attribute(it)->info()=19; }

  // 3-Sew the two cubes along one facet.
  lcc.sew<3>(d1->beta(1)->beta(1)->beta(2), d2->beta(2));

  // Barycentric triangulation of the facet between the two cubes.
  Dart_handle d3=lcc.insert_barycenter_in_cell<2>(d2->beta(2));

  // Set the color of the new vertex to 5.
  LCC_3::vertex_attribute(d3)->info()=5;
```

```
  // Display all the vertices of the map.
  for (LCC_3::Vertex_attribute_range::iterator
         it=lcc.vertex_attributes().begin(),
         itend=lcc.vertex_attributes().end();
       it!=itend; ++it)
  {
    std::cout<<"point: "<<it->point()<<", "<<"color: "<<it->info()
             <<std::endl;
  }

  return EXIT_SUCCESS;
}
```

File: examples/Linear_cell_complex/linear_cell_complex_3_with_colored_vertices.cpp

The output is:

```
point: -1 1 1, color: 10
point: -1 0 1, color: 10
point: -2 0 1, color: 1
point: -2 1 1, color: 1
point: -2 1 0, color: 1
point: -1 1 0, color: 10
point: -1 0 0, color: 10
point: -2 0 0, color: 1
point: 1 1 1, color: 19
point: 1 0 1, color: 19
point: -1 0.5 0.5, color: 5
point: 1 1 0, color: 19
point: 1 0 0, color: 19
```

Before applying the sew operation, the eight vertices of the first cube are colored by 1, and the eight vertices of the second cube by 19. After the sew operation, there are eight vertices which are merged two by two, and due to the average functor, the color of the four resulting vertices is now 10. Then we insert a vertex in the center of the common 2-cell between the two cubes. The coordinates of this vertex are initialized with the barycenter of the 2-cell (-1,0.5,0.5), and its color is not initialized by the method, thus we set its color manually by using the result of *insert_barycenter_in_cell*<2> which is a dart incident to the new vertex.

28.5 Design and Implementation History

This package was developed by Guillaume Damiand, with the help of Andreas Fabri, Sébastien Loriot and Laurent Rineau. Monique Teillaud and Bernd Gärtner contributed to the manual.

Linear Cell Complex Reference Manual

28.6 Classified Reference Pages

28.6.1 Concepts

28.6.2 Classes

28.6.3 Global Functions

Constructions for Linear cell complex

Operations for Linear cell complex

28.7 Alphabetical List of Reference Pages

LinearCellComplexTraits

Required types and functors for the LinearCellComplexTraits concept. This geometric traits concept is used in the *Linear_cell_complex* class.

Constants

static unsigned int	*ambient_dimension;*	The ambient dimension, must be >1.

Types

LinearCellComplexTraits:: FT	a number type that is a model of FieldNumberType.
LinearCellComplexTraits:: Point	point type.
LinearCellComplexTraits:: Vector	vector type.

Constructions

LinearCellComplexTraits:: Construct_translated_point

Functor that provides *Point operator() (const Point& p, const Vector& v)*, which constructs the translation of point *p* by vector *v*, and *Point operator() (const CGAL::Origin&, const Vector& v)*, which constructs the translation of a point at the origin by vector *v* (used in *Linear_cell_complex::barycenter*).

LinearCellComplexTraits:: Construct_vector

Functor that provides *Vector operator() (const Point& p1, const Point& p2)* which constructs a vector as the difference of points *p2-p1*, and *Vector operator() (const CGAL::Origin&, const Point& p)* which constructs a vector as the difference of point *p* and a point at the origin (used in *Linear_cell_complex::barycenter* and *CGAL::import_from_plane_graph*).

LinearCellComplexTraits:: Construct_sum_of_vectors

Functor that provides *Vector operator() (const Vector& v1, const Vector& v2)* which constructs a vector as the sum of vectors *v1+v2* (used in *Linear_cell_complex::barycenter*, *CGAL::compute_normal_of_cell_0* and *CGAL::compute_normal_of_cell_2*).

LinearCellComplexTraits:: Construct_scaled_vector

Functor that provides *Vector operator() (const Vector& v, FT scale)* which constructs a vector equal to vector *v* scaled by *scale* factor (used in *Linear_cell_complex::barycenter* , *CGAL::compute_normal_of_cell_0* and *CGAL::compute_normal_of_cell_2*).

LinearCellComplexTraits:: Construct_midpoint

Functor that provides *Point operator() (const Point& p1, const Point& p2)* which constructs the midpoint of points *p1* and *p2* (used in *Linear_cell_complex::barycenter*).

If *ambient_dimension==2*

LinearCellComplexTraits:: Direction_2	a model of *Direction_2*.
LinearCellComplexTraits:: Construct_direction_2	
	a model of *ConstructDirection_2* (used in *CGAL::import_from_plane_graph*).

If *ambient_dimension==3*

LinearCellComplexTraits:: Construct_normal_3	
	a model of *ConstructNormal_3* (used in *CGAL::compute_normal_of_cell_2*).
LinearCellComplexTraits:: Collinear_3	a model of *Collinear_3* (used in *CGAL::compute_normal_of_cell_2*).

Has Models

CGAL::Linear_cell_complex_traits<d,K> . . . . page 1983.

See Also

CGAL::Linear_cell_complex<d,d2,LCCTraits,Items,Alloc> . . . . page 1975

LinearCellComplexItems

Definition

The concept LinearCellComplexItems refines the concept of *CombinatorialMapItems* by adding the requirement that 0-attributes are enabled, and associated with attributes that are models of the *CellAttributeWithPoint* concept.

Refines

CombinatorialMapItems . . . . page 1934

Requirements

The first type in *Attributes* must be a model of the *CellAttributeWithPoint* concept.

Has Models

CGAL::Linear_cell_complex_min_items<d> . . . . page 1982

See Also

CGAL::Linear_cell_complex<d,d2,LCCTraits,Items,Alloc> . . . . page 1975
CellAttributeWithPoint . . . . page 1974
CGAL::Dart<d,CMap> . . . . page 1938

Concept

CellAttributeWithPoint

Definition

The concept CellAttributeWithPoint is a refinement of the *CellAttribute* concept, to represent a cell attribute containing a point.

Refines

CellAttribute . . . page 1932

Types

CellAttributeWithPoint:: Point	Type of the used point.

Creation

CellAttributeWithPoint cawp;	Default constructor.
CellAttributeWithPoint cawp(Point apoint);	Constructor initializing the point of *cawp* by the copy contructor *Point(apoint)*.
CellAttributeWithPoint cawp(Point apoint, Info info);	Constructor initializing the point of *cawp* by the copy contructor *Point(apoint)* and initializing the information of *cawp* by the copy contructor *Info(info)*. Defined only if *Info* is different from *void*.

Access Member Functions

Point&	*cawp.point()*	Returns the point of *cawp*.
Point	*cawp.point() const*	Returns the point of *cawp*, when *cawp* is const.

Has Models

CGAL::Cell_attribute_with_point<LCC,Info_,Tag,OnMerge,OnSplit> . . . page 1984

See Also

LinearCellComplexItems . . . page 1973

CGAL::Linear_cell_complex<d,d2,LCCTraits,Items,Alloc>

Class

#include <CGAL/Linear_cell_complex.h>

Definition

The class *Linear_cell_complex<d,d2,LCCTraits,Items,Alloc>* represents a linear cell complex in dimension *d*, in an ambient space of dimension *d2*. This is a model of the concept of *CombinatorialMap* adding a requirement to ensure that each vertex of the map is associated with a model of *CellAttributeWithPoint*.

Is Model for the Concepts

CombinatorialMap . . . . page 1919

Inherits From

CGAL::Combinatorial_map<d,Items,Alloc> . . . . page 1936

Parameters

d an integer for the dimension of the combinatorial map,
d2 an integer for the dimension of the ambient space,
LCCTraits must be a model of the *LinearCellComplexTraits* concept, satisfying *LCCTraits::ambiant_dimension==d2*,
Items must be a model of the *LinearCellComplexItems* concept,
Alloc has to match the standard allocator requirements.

There are four default template arguments: *d2* is equal to *d*, *LCCTraits* is equal to *CGAL::Linear_cell_complex_traits<d2>*, *Items* is equal to *CGAL::Linear_cell_complex_min_items<d>* and *Alloc* is *CGAL_ALLOCATOR(int)*.

advanced

Note that there is an additional, and undocumented, template parameter *CMap* for *Linear_cell_complex<d,d2,LCCTraits,Items,Alloc,CMap>* allowing to inherit from any model of the *CombinatorialMap* concept.

advanced

Creation

Linear_cell_complex<d,d2,LCCTraits,Items,Alloc> lcc;

Constants

static unsigned int *ambient_dimension = d2;*

must be >1.

Types

typedef Linear_cell_complex<d,d2,LCCTraits,Items,Alloc>	*Self;*
typedef Items::Dart_wrapper<Self>::Dart	*Dart;*

The type of dart, must satisfy *Dart::dimension==d*.

typedef LCCTraits	*Traits;*
typedef Items	*Items;*
typedef Alloc	*Alloc;*

typedef Traits::FT	*FT;*
typedef Traits::Point	*Point;*
typedef Traits::Vector	*Vector;*

Linear_cell_complex<d,d2,LCCTraits,Items,Alloc>:: Vertex_attribute

Type of 0-attributes, a model of *CellAttributeWithPoint* concept (a shortcut for *Attribute_type_d<0>::type*).

Linear_cell_complex<d,d2,LCCTraits,Items,Alloc>:: Vertex_attribute_handle

Handle through 0-attributes (a shortcut for *Attribute_handle_type_d<0>::type*).

Linear_cell_complex<d,d2,LCCTraits,Items,Alloc>:: Vertex_attribute_const_handle

Const handle through 0-attributes (a shortcut for *Attribute_const_handle_type_d<0>::type*).

Linear_cell_complex<d,d2,LCCTraits,Items,Alloc>:: Vertex_attribute_range

Range of all the 0-attributes, a model of the *Range* concept (a shortcut for *Attribute_range_d<0>::type*). Iterator inner type is bidirectional iterator and value type is *Vertex_attribute*.

Linear_cell_complex<d,d2,LCCTraits,Items,Alloc>:: Vertex_attribute_const_range

Const range of all the 0-attributes, a model of the *ConstRange* concept a shortcut for *Attribute_const_range_d<0>::type*). Iterator inner type is bidirectional iterator and value type is *Vertex_attribute*.

Range Access Member Functions

Vertex_attribute_range& *lcc.vertex_attributes()*

Returns a range of all the 0-attributes in *lcc* (a shortcut for *attributes<0>()*).

Vertex_attribute_const_range& *lcc.vertex_attributes() const*

Returns a const range of all the 0-attributes in *lcc* (a shortcut for *attributes<0>() const*).

Access Member Functions

bool *lcc.is_valid() const* Returns true iff *lcc* is valid.

A linear cell complex *lcc* is valid if it is a valid combinatorial map, and if for each dart handle *dh* such that **dh*∈*lcc.darts()*: *dh->attribute<0>()!=NULL.*

size_type *lcc.number_of_vertex_attributes() const*

Returns the number of 0-attributes in *lcc* (a shortcut for *number_of_attributes<0>()*).

Static Member Functions

static Vertex_attribute_handle *vertex_attribute(Dart_handle dh)*

Returns the 0-attribute associated with *dh*.

static Vertex_attribute_const_handle *vertex_attribute(Dart_const_handle dh)*

Returns the 0-attribute associated with *dh*, when *dh* is const.

static Point& *point(Dart_handle dh)*

Returns the point in the 0-attribute associated with *dh*.

static Point *point(Dart_const_handle dh)*

Returns the point in the 0-attribute associated with *dh*, when *dh* is const.

Modifiers

Dart_handle *lcc.create_dart(Vertex_attribute_handle vh)*

Creates a new dart in *lcc*, sets its associated 0-attribute to *vh* and returns the corresponding handle.
Precondition: **vh*∈*lcc.vertex_attributes()*.

Dart_handle *lcc.create_dart(Point apoint)*

Creates a new dart in *lcc*, creates a new 0-attribute initialized with *apoint*, sets the associated 0-attribute of the new dart to this new 0-attribute, and returns the corresponding handle.

template<typename T1>
Vertex_attribute_handle *lcc.create_vertex_attribute(T1 t1)*

Creates a new 0-attribute in *lcc*, and returns the corresponding handle (a shortcut for *create_attribute<0> (t1)*). Calls the constructor of *Vertex_attribute* having *T1* as parameter. Overloads of this member function are defined that take from zero to nine arguments. With zero argument, *create_vertex_attribute()* creates a new 0-attribute by using the default constructor.

void *lcc.erase_vertex_attribute(Vertex_attribute_handle vh)*

Removes the 0-attribute pointed to by *vh* from *lcc* (a shortcut for *erase_attribute<0>(vh)*).
Precondition: **vh*$\in$*lcc.vertex_attributes()*.

void *lcc.set_vertex_attribute(Dart_handle dh, Vertex_attribute_handle vh)*

Associates the 0-attribute of all the darts of the 0-cell containing *dh* to *vh* (a shortcut for *set_attribute<0> (dh,vh)*).
Precondition: **dh*$\in$*lcc.darts()* and **vh*$\in$*lcc.vertex_attributes()*.

Operations

template<unsigned int i>
Point *lcc.barycenter(Dart_const_handle dh) const*

Returns the barycenter of the *i*-cell containing *dh*.
Precondition: $1 \leq i \leq$ *dimension* and **dh*$\in$*lcc.darts()*.

template <unsigned int i>
Dart_handle *lcc.insert_point_in_cell(Dart_handle dh, Point p)*

Inserts a point, copy of *p*, in the *i*-cell containing *dh*. Returns a handle on one dart of this cell.
Precondition: $1 \leq$ *dimension* ≤ 2 and **dh*$\in$*lcc.darts()*.
If *i*-attributes are non void, *Attribute_type<i> ::type::On_split*(*a*,*a'*) is called, with *a* the original *i*-attribute associated with *dh* and *a'* each new *i*-attribute created during the operation.

template <unsigned int i>

Dart_handle *lcc.insert_barycenter_in_cell(Dart_handle dh)*

Inserts a point in the barycenter of the *i*-cell containing *dh*. Returns a handle on one dart of this cell.
Precondition: $1 \leq$ *dimension* ≤ 2 and **dh*$\in$*lcc.darts()*.
If *i*-attributes are non void, *Attribute_type<i> ::type::On_split*(*a*,*a'*) is called, with *a* the original *i*-attribute associated with *dh* and *a'* each new *i*-attribute created during the operation.

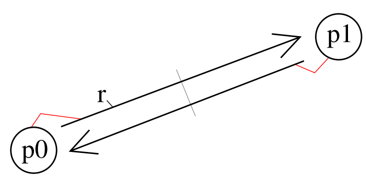
Dart_handle *lcc.insert_dangling_cell_1_in_cell_2(Dart_handle dh, Point p)*

Inserts a 1-cell in the 2-cell containing *dh*, the 1-cell being attached only by one of its vertex to the 0-cell containing *dh*. The second vertex is associated with a new 0-attribute containing a copy of *p* as point. Returns a handle on one dart belonging to the new 0-cell.
Precondition: *dimension* ≥ 2 and **dh*$\in$*lcc.darts()*.

Constructions

Dart_handle *lcc.make_segment(Point p0, Point p1)*

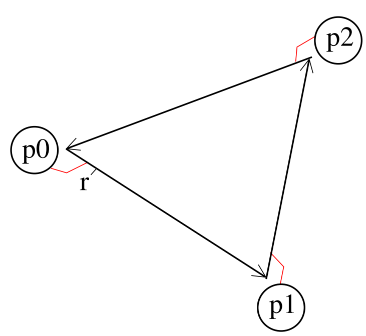
Creates an isolated segment in *lcc* (two darts linked by β_2) having *p0*, *p1* as points. Returns a handle on the dart associated with *p0*.
Precondition: *dimension* ≥ 2.



Example of *r=lcc.make_segment(p0,p1)*.

Dart_handle *lcc.make_triangle(Point p0, Point p1, Point p2)*

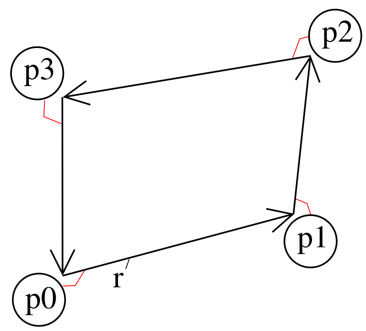
Creates an isolated triangle in *lcc* having *p0*, *p1*, *p2* as points. Returns a handle on the dart associated with *p0*.
Precondition: *dimension* ≥ 1.



Example of *r=lcc.make_triangle(p0,p1,p2)*.

Dart_handle *lcc.make_quadrangle(Point p0, Point p1, Point p2, Point p3)*

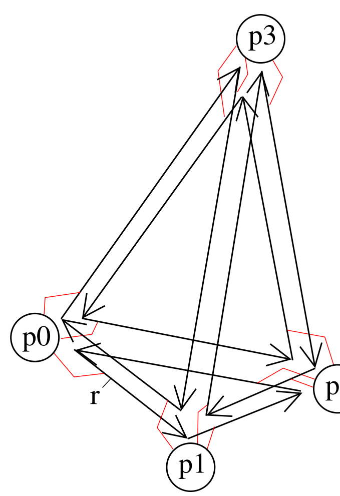
Creates an isolated quadrangle in *lcc* having *p0*, *p1*, *p2*, *p3* as points. Returns a handle on the dart associated with *p0*.
Precondition: *dimension*≥1.



Example of *r=lcc.make_quadrangle(p0,p1,p2,p3)*.

Dart_handle *lcc.make_tetrahedron(Point p0, Point p1, Point p2, Point p3)*

Creates an isolated tetrahedron in *lcc* having *p0*, *p1*,*p2*,*p3* as points. Returns a handle on the dart associated with *p0* and belonging to the 2-cell having *p0*, *p1*, *p2* as points.
Precondition: *dimension*≥2.

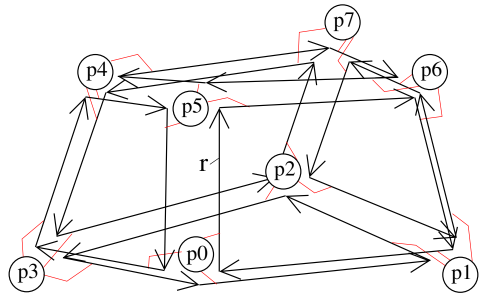


Example of *r=lcc.make_tetrahedron(p0,p1,p2,p3)*.

Dart_handle *make_hexahedron(Point p0,*
Point p1,
Point p2,
Point p3,
Point p4,
Point p5,
Point p6,

Point p7)

Creates an isolated hexahedron in *lcc* having *p0*, *p1*, *p2*, *p3*, *p4*, *p5*, *p6*, *p7* as points. Returns a handle on the dart associated with *p0* and belonging to the 2-cell having *p0*, *p5*, *p6*, *p1* as points.
Precondition: *dimension*≥2.



Example of *r=lcc.make_hexahedron(p0,p1,p2,p3,p4,p5,p6,p7)*.

See Also

CombinatorialMap . . . page 1919
CGAL::Combinatorial_map<d,Items,Alloc> . . . page 1936
Dart . . . page 1930
LinearCellComplexItems . . . page 1973
CGAL::Linear_cell_complex_min_items<d> . . . page 1982
LinearCellComplexTraits . . . page 1971
CGAL::Linear_cell_complex_traits<d,K> . . . page 1983

Class

CGAL::Linear_cell_complex_min_items<d>

#include <CGAL/Linear_cell_complex_min_items.h>

Definition

The class *Linear_cell_complex_min_items<d>* defines the type of darts, which is a *Dart_wrapper::Dart<d,LCC>* , and the traits class used. In this class, 0-attributes are enabled and associated with *Cell_attribute_with_point*.

Is Model for the Concepts

LinearCellComplexItems . . . . page 1973

Parameters

d the dimension of the combinatorial map.

Example

The following example shows one implementation of the *Linear_cell_complex_min_items<d>* class.

```
template <unsigned int d>
struct Linear_cell_complex_min_items
{
  template <class LCC>
  struct Dart_wrapper
  {
    typedef CGAL::Dart<d, LCC> Dart;

    typedef CGAL::Cell_attribute_with_point<LCC> Vertex_attrib;
    typedef CGAL::cpp0x::tuple<Vertex_attrib> Attributes;
  };
};
```

See Also

CGAL::Linear_cell_complex<d,d2,LCCTraits,Items,Alloc> . . . . page 1975
CGAL::Dart<d,CMap> . . . . page 1938

CGAL::Linear_cell_complex_traits<d,K>

#include <CGAL/Linear_cell_complex_traits.h>

Class

Definition

This geometric traits concept is used in the *Linear_cell_complex* class. It can take as parameter any model of the concept *Kernel* (for example any CGAL kernel), and defines inner types and functors corresponding to the given dimension.

Is Model for the Concepts

LinearCellComplexTraits . . . . page 1971

Inherits From

K.

Parameters

d the dimension of the kernel,
K a model of the concept *Kernel* if *d==2* or *d==3*; a model of the concept *Kernel_d* otherwise.

There is a default template arguments for *K* which is *CGAL::Exact_predicates_inexact_constructions_kernel* if *d* is 2 or 3, and is *CGAL::Cartesian_d<double>* otherwise.

Note that the default argument used for *K* when *d*>3 does not use exact predicates because operations that use predicates are only defined in 2D and 3D.

Constants

static unsigned int *ambient_dimension = d;*

See Also

CGAL::Linear_cell_complex<d,d2,LCCTraits,Items,Alloc> . . . . page 1975

Class

CGAL::Cell_attribute_with_point<LCC,Info_,Tag,OnMerge,OnSplit>

#include <CGAL/Cell_attribute_with_point.h>

Definition

The class *Cell_attribute_with_point<LCC,Info_,Tag,OnMerge,OnSplit>* represents an attribute containing a point and containing an information when *Info_* is different from void. This class can typically be used to associate a point to each 0-cell of a combinatorial map.

Is Model for the Concepts

CellAttributeWithPoint . . . . page 1974

Inherits From

CGAL::Cell_attribute<CMap,Info_,Tag,OnMerge,OnSplit> . . . . page 1939

Parameters

LCC must be an instantiation of *Linear_cell_complex* class,
*Info_* is the type of the information contained in the attribute, *void* for no information,
Tag is *Tag_true* to enable the storage of a *Dart_handle* of the associated cell, *Tag_false* otherwise,
OnMerge is a functor called when two attributes are merged,
OnSplit is a functor called when one attribute is split in two.

By default, *OnMerge* and *OnSplit* are equal to *Null_functor*; *Tag* is equal to *Tag_true*; and *Info_* is equal to *void*.

Types

typedef LCC::Point	*Point;*
typedef LCC::Dart_handle	*Dart_handle;*
typedef LCC::Dart_const_handle	*Dart_const_handle;*

See Also

CGAL::Linear_cell_complex<d,d2,LCCTraits,Items,Alloc> . . . . page 1975
CGAL::Linear_cell_complex_min_items<d> . . . . page 1982
CGAL::Cell_attribute<CMap,Info_,Tag,OnMerge,OnSplit> . . . . page 1939

CGAL::import_from_plane_graph<LCC>

#include <CGAL/Linear_cell_complex_constructors.h>

template<class LCC>
typename LCC::Dart_handle

import_from_plane_graph(LCC& lcc, std::istream& ais)

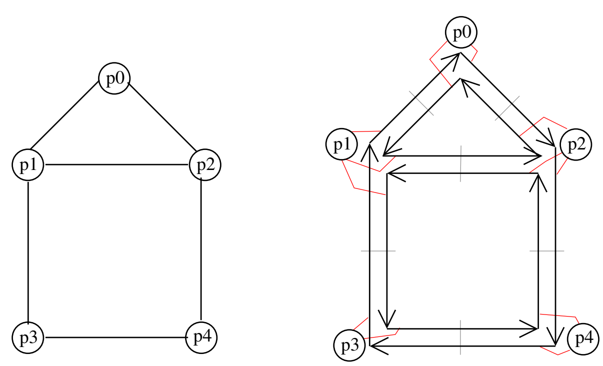
Imports an embedded plane graph read from *ais* into *lcc*. Objects are added in *lcc*, existing darts are not modified. Returns a dart created during the import.
Precondition: *LCC::dimension*≥2 and *LCC::ambient_dimension*==2.

File format

The file format must be the following. First the number of vertices and the number of edges of the planar graph. Then, for each vertex of the planar graph, the coordinates of the i^{th} vertex (two numbers for x and y coordinates). The first vertex index is 0. Then for each edge of the planar graph, the two indices of the two vertices (two numbers between 0 and the number of vertices minus 1).

Here a small example:

```
5 6
1.0 3.0    0.0 2.0    2.0 2.0    0.0 0.0    2.0 0.0
0 1   0 2   1 2   1 3   2 4   3 4
```



Example of *import_graph* reading the above file as istream.
Left: A planar graph embedded in the plane with *P0*=(1.0,3.0), *P1*=(0.0,2.0), *P2*=(2.0,2.0), *P3*=(0.0,0.0), *P4*=(2.0,0.0). **Right**: the 2D linear cell complex reconstructed.

See Also

Function

CGAL::import_from_triangulation_3<LCC,Triangulation>

#include <CGAL/Linear_cell_complex_constructors.h>

template <class LCC,class Triangulation_>
typename LCC::Dart_handle

import_from_triangulation_3(LCC& lcc, Triangulation_ atr)

Imports *atr* (a *Triangulation_3*) into *lcc*. Objects are added in *lcc*, existing darts are not modified. Returns a dart created during the import.
Precondition: *LCC::dimension*≥3 and *LCC::ambient_dimension*==3.

See Also

CGAL::import_from_plane_graph<LCC> . . . . page 1985
CGAL::import_from_polyhedron_3<LCC,Polyhedron> . . . . page 1987

CGAL::import_from_polyhedron_3<LCC,Polyhedron>

Function

#include <CGAL/Linear_cell_complex_constructors.h>

template<class LCC,class Polyhedron>
typename LCC::Dart_handle

import_from_polyhedron_3(LCC& lcc, Polyhedron &apoly)

Imports *apoly* (a *Polyhedron_3*) into *lcc*. Objects are added in *lcc*, existing darts are not modified. Returns a dart created during the import.
Precondition: *LCC::dimension*$\geq$2 and *LCC::ambient_dimension*==3.

See Also

CGAL::import_from_plane_graph<LCC> ... page 1985
CGAL::import_from_triangulation_3<LCC,Triangulation> ... page 1986

Function

CGAL::compute_normal_of_cell_0<LCC>

#include <CGAL/Linear_cell_complex_operations.h>
template <class LCC>
typename LCC::Vector

compute_normal_of_cell_0(LCC lcc, typename LCC::Dart_const_handle dh)

Returns the normal vector of the 0-cell containing *dh*; i.e. the average of all the normal vectors of the 2-cells incident to the 0-cell containing *dh*.
Precondition: *LCC::ambient_dimension*==3 and **dh*$\in$*lcc.darts()*.

See Also

CGAL::compute_normal_of_cell_2<LCC> . . . . page 1989

CGAL::compute_normal_of_cell_2<LCC>

#include <CGAL/Linear_cell_complex_operations.h>
template <class LCC>
typename LCC::Vector

compute_normal_of_cell_2(LCC lcc, typename LCC::Dart_const_handle dh)

Returns the normal vector of the 2-cell containing *dh*.
Precondition: *LCC::ambient_dimension*==3 and **dh*∈*lcc.darts()*.

Function

See Also

CGAL::compute_normal_of_cell_0<LCC> . . . . page 1988

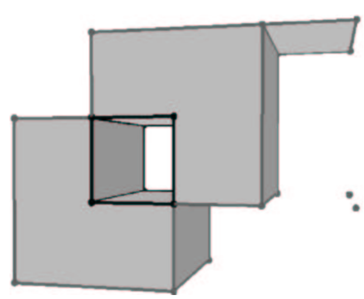
Chapter 29

3D Boolean Operations on Nef Polyhedra

Peter Hachenberger and Lutz Kettner

Contents

29.1 Introduction



In solid modeling, two major representation schemes are used: *constructive solid geometry* (CSG) and *boundary representations* (B-rep). Both have inherent strengths and weaknesses, see [Hof89c] for a discussion.

In CSG a solid is represented as a set-theoretic boolean combination of primitive solid objects, such as blocks, prisms, cylinders, or toruses. The boolean operations are not evaluated, instead, objects are represented implicitly with a tree structure; leaves represent primitive objects and interior nodes represent boolean operations or rigid motions, e.g., translation and rotation. Algorithms on such a CSG-tree first evaluate properties on the primitive objects and propagate the results using the tree structure.

A B-rep describes the incidence structure and the geometric properties of all lower-dimensional features of the boundary of a solid. Surfaces are oriented to decide between the interior and exterior of a solid.

The class of representable objects in a CSG is usually limited by the choice of the primitive solids. A B-rep is usually limited by the choice for the geometry of the supporting curves for edges and the supporting surfaces for surface patches, and, in addition, the connectivity structure that is allowed. In particular, a B-rep is not always closed under boolean set operations. As an example, the class of orientable 2-manifold objects is a popular and well understood class of surfaces commonly used for B-reps. They can be represented and manipulated efficiently, the data structures are compact in storage size, and many algorithms are simple. On the other side, this object class is not closed under boolean set operations, as many examples can illustrate, such as the Figure shown above that can be generated using boolean set operations on cubes. The vertices bounding the tunnel, or the edge connecting the "roof" with the cube are non-manifold situations.

In our implementation of Nef polyhedra in 3D, we offer a B-rep data structure that is closed under boolean operations and with all their generality. Starting from halfspaces (and also directly from oriented 2-manifolds), we can work with set union, set intersection, set difference, set complement, interior, exterior, boundary, closure, and regularization operations (see Section 29.4 for an explaination of regularized set operations). In essence, we can evaluate a CSG-tree with halfspaces as primitives and convert it into a B-rep representation.

In fact, we work with two data structures; one that represents the local neighborhoods of vertices, which is in itself already a complete description, and a data structure that connects these neighborhoods up to a global data structure with edges, facets, and volumes. We offer a rich interface to investigate these data structures, their different elements and their connectivity. We provide affine (rigid) transformations and a point location query operation. We have a custom file format for storing and reading Nef polyhedra from files. We offer a simple OpenGL-based visualizer for debugging and illustrations.

29.2 Definition

The theory of Nef polyhedra has been developed for arbitrary dimensions. The class *CGAL::Nef_polyhedron_3* implements a boundary representation for the 3-dimensional case.

Definition: A *Nef-polyhedron* in dimension d is a point set $P \subseteq \mathbb{R}^d$ generated from a finite number of open halfspaces by set complement and set intersection operations.

Set union, difference and symmetric difference can be reduced to intersection and complement. Set complement changes between open and closed halfspaces, thus the topological operations *boundary*, *interior*, *exterior*, *closure* and *regularization* are also in the modeling space of Nef polyhedra.

A face of a Nef polyhedron is defined as an equivalence class of *local pyramids* that are a characterization of the local space around a point.

Definition: A point set $K \subseteq \mathbb{R}^d$ is called a *cone with apex* 0, if $K = \mathbb{R}^+ K$ (i.e., $\forall p \in K, \forall \lambda > 0 : \lambda p \in K$) and it is called a *cone with apex* x, $x \in \mathbb{R}^d$, if $K = x + \mathbb{R}^+(K - x)$. A cone K is called a *pyramid* if K is a polyhedron.

Now let $P \in \mathbb{R}^d$ be a polyhedron and $x \in \mathbb{R}^d$. There is a neighborhood $U_0(x)$ of x such that the pyramid $Q := x + \mathbb{R}^+((P \cap U(x)) - x)$ is the same for all neighborhoods $U(x) \subseteq U_0(x)$. Q is called the *local pyramid* of P in x and denoted $\mathrm{Pyr}_P(x)$.

Definition: Let $P \in \mathbb{R}^d$ be a polyhedron and $x, y \in \mathbb{R}^d$ be two points. We define an equivalence relation $x \sim y$ iff $\mathrm{Pyr}_P(x) = \mathrm{Pyr}_P(y)$. The equivalence classes of $\sim$ are the *faces* of P. The dimension of a face s is the dimension of its affine hull, $\dim s := \dim \mathrm{aff}\, s$.

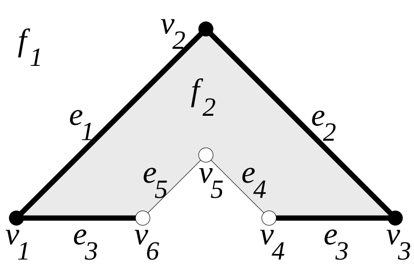
In other words, a *face* s of P is a maximal non-empty subset of $\mathbb{R}^d$ such that all of its points have the same local pyramid Q denoted $\mathrm{Pyr}_P(s)$. This definition of a face partitions $\mathbb{R}^d$ into faces of different dimension. A face s is either a subset of P, or disjoint from P. We use this later in our data structure and store a selection mark in each face indicating its set membership.

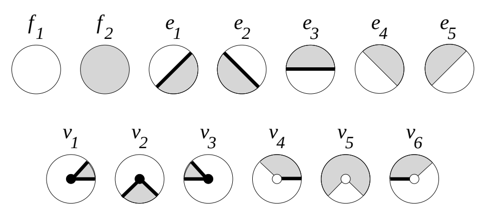
Faces do not have to be connected. There are only two full-dimensional faces possible, one whose local pyramid is the space $\mathbb{R}^d$ itself and the other with the empty set as a local pyramid. All lower-dimensional faces form the *boundary* of the polyhedron. As usual, we call zero-dimensional faces *vertices* and one-dimensional faces *edges*. In the case of polyhedra in space we call two-dimensional faces *facets* and the full-dimensional faces *volumes*. Faces are *relative open* sets, e.g., an edge does not contain its end-vertices.

We illustrate the definitions with an example in the plane. Given the closed halfspaces

$$h_1 : y \geq 0, \qquad h_2 : x - y \geq 0, \qquad h_3 : x + y \leq 3, \qquad h_4 : x - y \geq 1, \qquad h_5 : x + y \leq 2,$$

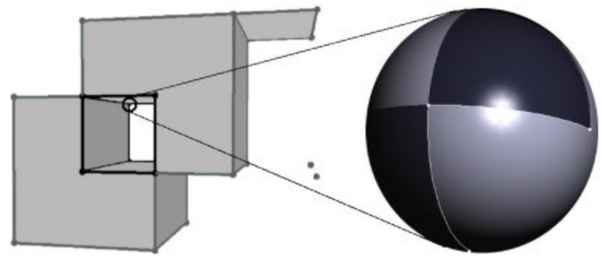
we define our polyhedron $P := (h_1 \cap h_2 \cap h_3) - (h_4 \cap h_5)$.



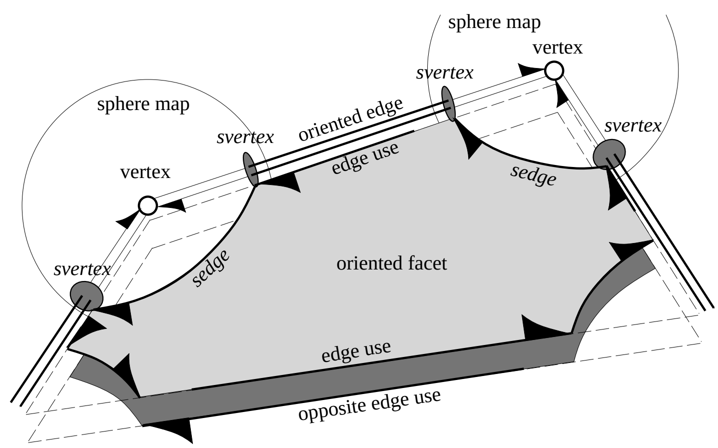


The left figure illustrates the polyhedron with its partially closed and partially open boundary, i.e., vertex v_4, v_5, v_6, and edges e_4 and e_5 are not part of P. The local pyramids for the faces are $\mathrm{Pyr}_P(f_1) = \emptyset$ and $\mathrm{Pyr}_P(f_2) = \mathbb{R}^2$. Examples for the local pyramids of edges are the closed halfspace h_2 for the edge e_1, $\mathrm{Pyr}_P(e_1) = h_2$, and the open halfspace that is the complement of h_4 for the edge e_5, $\mathrm{Pyr}_P(e_5) = \{(x,y) | x - y < 1\}$. The edge e_3 consists actually of two disconnected parts, both with the same local pyramid $\mathrm{Pyr}_P(e_3) = h_1$. In our data structure, we will represent the two connected components of the edge e_3 separately. The figure on the right lists all local pyramids for this example.

The local pyramids of each vertex are represented by conceptually intersecting the local neighborhood with a small ε-sphere. This intersection forms a planar map on the sphere (see next two figures), which together with the set-selection mark for each item (i.e. vertices, edges, loops and faces) forms a two-dimensional Nef polyhedron embedded in the sphere. We add the set-selection mark for the vertex and call the resulting structure the *sphere map* of the vertex. We use the prefix s to distinguish the elements of the sphere map from the three-dimensional elements. See Chapter 21 for further details.



Having sphere maps for all vertices of our polyhedron is a sufficient but not easily accessible representation of the polyhedron. We enrich the data structure with more explicit representations of all the faces and incidences between them.



We depart slightly from the definition of faces in a Nef polyhedron; we represent the connected components of a face individually and do not implement additional bookkeeping to recover the original faces (e.g., all edges on a common supporting line with the same local pyramid) as this is not needed in our algorithms. We discuss features in the increasing order of dimension.

edges: We store two oppositely oriented edges for each edge and have a pointer from one oriented edge to its opposite edge. Such an oriented edge can be identified with an *svertex* in a sphere map; it remains to link one *svertex* with the corresponding opposite *svertex* in the other sphere map.

edge uses: An edge can have many incident facets (non-manifold situation). We introduce two oppositely oriented edge-uses for each incident facet; one for each orientation of the facet. An edge-use points to its corresponding oriented edge and to its oriented facet. We can identify an edge-use with an oriented *sedge* in the sphere map, or, in the special case also with an *sloop*. Without mentioning it explicitly in the remainder, all references to *sedge* can also refer to *sloop*.

facets: We store oriented facets as boundary cycles of oriented edge-uses. We have a distinguished outer boundary cycle and several (or maybe none) inner boundary cycles representing holes in the facet. Boundary cycles are linked in one direction. We can access the other traversal direction when we switch to the oppositely oriented facet, i.e., by using the opposite edge-use.

shells: The volume boundary decomposes into different connected components, the *shells*. A shell consists of a connected set of facets, edges, and vertices incident to this volume. Facets around an edge form a radial order that is captured in the radial order of *sedges* around an *svertex* in the sphere map. Using this information, we can trace a shell from one entry element with a graph search. We offer this graph traversal (to the user) in a visitor design pattern.

volumes: A volume is defined by a set of shells, one outer shell containing the volume and several (or maybe none) inner shells separating voids which are excluded from the volume.

For each face we store a label, e.g., a set-selection mark, which indicates whether the face is part of the solid or if it is excluded. We call the resulting data structure *Selective Nef Complex*, *SNC* for short [GHH+03]. However, in CGAL we identify the names and call the *SNC* data structure *CGAL::Nef_polyhedron_3*.

29.3 Infimaximal Box

We call a Nef polyhedron *bounded* if its boundary is bounded, i.e., finite, and *unbounded* otherwise. Note that unbounded point sets can have a bounded boundary, for example, the complement of a cube has an unbounded outer volume, but its boundary remains bounded.

Using a boundary representation, it is convenient (conceptually and in our implementation) to consider bounded Nef polyhedra only. Bounded Nef polyhedra are also closed under boolean set operations. However, one needs to start with bounded primitives; the conceptually nice halfspaces cannot be used. Instead, we offer a construction from oriented 2-manifolds represented in a *CGAL::Polyhedron_3*, see Section 29.5.4 below.

In order to handle unbounded Nef polyhedra conceptually in the same way as we handle bounded Nef polyhedra, we intersect them with a bounding cubical volume of size $[-R,R]^3$, where R is a symbolical unspecified value, which is finite but larger than all coordinate values that may occur in the bounded part of the polyhedron. As a result, each Nef polyhedron becomes bounded. We call the boundary of the bounding volume the *infimaximal box* [SM00].

We clip lines and rays at the infimaximal box. The intersection points with the infimaximal box are called *non-standard points*, which are points whose coordinates are $-R$ or R in at least one dimension, and linear functions $f(R)$ for the other dimensions. Such extended points (and developed from there also extended segments etc) are provided in CGAL with extended kernels—*CGAL::Extended_cartesian* and *CGAL::Extended_homogeneous*. They are regular CGAL kernels with a polynomial type as coordinate number type.

As long as an extended kernel is used, the full functionality provided by the *CGAL::Nef_polyhedron_3* class is available. If a kernel that does not use polynomials to represent coordinates is used, it is not possible to create or load unbounded Nef polyhedra, but all other operations work as expected. We provided both possibilities, since the restriction to bounded Nef polyhedra improves considerably space requirements (plain number type instead of polynomial), and runtime performance.

29.4 Regularized Set Operations

Since manifolds are not closed under boolean operations, Requicha proposes to use *regularized set operations* [KM76, Req80]. A set is *regular*, if it equals the closure of its interior. A regularized set operation is defined as the standard set operation followed by a regularization of the result. Regularized sets are closed under regularized set operations.

Regularized set operations are important since they simplify the class of solids to exclude lower dimensional features and the boundary belongs to the point set. These properties are considered to reflect the nature of physical solids more closely.

Regularized polyhedral sets are a subclass of Nef polyhedra. We provide the *regularization* operation as a shortcut for the consecutive execution of the *interior* and the *closure* operations.

29.5 Example Programs

The following example gives a first impression of how to instantiate and use *Nef_polyhedron_3*. We use the *CGAL::Cartesian* kernel. All Cartesian and homogeneous kernels of CGAL are suitable if the number type parameter follows the usual requirements of being a model of the *CGAL::FieldNumberType* concept for the Cartesian kernels, or the *CGAL::RingNumberType* concept for the homogeneous kernels, respectively. Note however, that in the current state, the Nef polyhedron works only with CGAL kernels. The implementation makes use of CGAL specific functions in kernel objects, and does not yet offer a designed interface to a clean kernel concept that could be offered by an external kernel as well.

The example creates two Nef polyhedra—*N0* is the empty set, while *N1* represents the full space, i.e., the set of all points in the 3-dimensional space. The assertion assures that the empty set is the complement of the full space.

```
#include <CGAL/Gmpz.h>
#include <CGAL/Homogeneous.h>
#include <CGAL/Nef_polyhedron_3.h>

typedef CGAL::Homogeneous<CGAL::Gmpz>  Kernel;
typedef CGAL::Nef_polyhedron_3<Kernel> Nef_polyhedron;

int main() {
  Nef_polyhedron N0(Nef_polyhedron::EMPTY);
  Nef_polyhedron N1(Nef_polyhedron::COMPLETE);

  CGAL_assertion (N0 == N1.complement());
  return 0;
}
```

File: examples/Nef_3/nef_3_simple.cpp

29.5.1 Construction and Comparison

This example shows the various constructors. We can create the empty set, which is also the default constructor, and the full space, i.e. all points of $\mathbb{R}^3$ belong to the polyhedron. We can create a halfspace defined by a plane bounding it. It is only available if an extended kernel is used. The halfspace constructor has a second parameter that specifies whether the defining plane belongs to the point set (*Nef_polyhedron::INCLUDED*) or not (*Nef_polyhedron::EXCLUDED*). The default value is *Nef_polyhedron::INCLUDED*. Additionally, we can create a *Nef_polyhedron_3* from a *Polyhedron_3*, see the Section 29.5.4 below.

We can compute the point sets of two Nef polyhedra for equality and proper subset relationships. We offer the usual comparison operators ==, !=, <=, >=, < and >.

Nef polyhedra have the important feature that a representation that is called the *reduced Würzburg structure* is unique, i.e., two point sets of Nef polyhedra are equal if and only if the representations are equal. The proof for the reduced Würzburg structure carries over to our representation and the comparison operators are therefore trivial to implement.

```
#include <CGAL/Gmpz.h>
#include <CGAL/Extended_homogeneous.h>
#include <CGAL/Nef_polyhedron_3.h>
```

```
typedef CGAL::Extended_homogeneous<CGAL::Gmpz>  Kernel;
typedef CGAL::Nef_polyhedron_3<Kernel>  Nef_polyhedron;
typedef Nef_polyhedron::Plane_3  Plane_3;

int main() {
  Nef_polyhedron N0;
  Nef_polyhedron N1(Nef_polyhedron::EMPTY);
  Nef_polyhedron N2(Nef_polyhedron::COMPLETE);
  Nef_polyhedron N3(Plane_3( 1, 2, 5,-1));
  Nef_polyhedron N4(Plane_3( 1, 2, 5,-1), Nef_polyhedron::INCLUDED);
  Nef_polyhedron N5(Plane_3( 1, 2, 5,-1), Nef_polyhedron::EXCLUDED);

  CGAL_assertion(N0 == N1);
  CGAL_assertion(N3 == N4);
  CGAL_assertion(N0 != N2);
  CGAL_assertion(N3 != N5);

  CGAL_assertion(N4 >= N5);
  CGAL_assertion(N5 <= N4);
  CGAL_assertion(N4 > N5);
  CGAL_assertion(N5 < N4);

  N5 = N5.closure();
  CGAL_assertion(N4 >= N5);
  CGAL_assertion(N4 <= N5);

  return 0;
}
```

File: examples/Nef_3/nef_3_construction.cpp

29.5.2 Point Set Operations

As explained in the introduction, Nef polyhedra are closed under all boolean set operations. The class *Nef_polyhedron_3* provides functions and operators for the most common ones: complement (*operator!*), union (*operator+*), difference (*operator-*), intersection (*operator**) and symmetric difference (*operator^*). Additionally, the operators *=, -=, *= and ^= are defined.

Nef_polyhedron_3 also provides the topological operations *interior()*, *closure()* and *boundary()*. With *interior()* one deselects all boundary items, with *boundary()* one deselects all volumes, and with *closure()* one selects all boundary items.

```
#include <CGAL/Gmpz.h>
#include <CGAL/Extended_homogeneous.h>
#include <CGAL/Nef_polyhedron_3.h>

typedef CGAL::Extended_homogeneous<CGAL::Gmpz>  Kernel;
typedef CGAL::Nef_polyhedron_3<Kernel>  Nef_polyhedron;
typedef Kernel::Plane_3  Plane_3;

int main() {
```

```
  Nef_polyhedron N1(Plane_3( 1, 0, 0,-1));
  Nef_polyhedron N2(Plane_3(-1, 0, 0,-1));
  Nef_polyhedron N3(Plane_3( 0, 1, 0,-1));
  Nef_polyhedron N4(Plane_3( 0,-1, 0,-1));
  Nef_polyhedron N5(Plane_3( 0, 0, 1,-1));
  Nef_polyhedron N6(Plane_3( 0, 0,-1,-1));

  Nef_polyhedron I1(!N1 + !N2);  // open slice in yz-plane
  Nef_polyhedron I2(N3 - !N4);   // closed slice in xz-plane
  Nef_polyhedron I3(N5 ^ N6);    // open slice in yz-plane
  Nef_polyhedron Cube1(I2 * !I1);
  Cube1 *= !I3;
  Nef_polyhedron Cube2 = N1 * N2 * N3 * N4 * N5 * N6;

  CGAL_assertion(Cube1 == Cube2);  // both are closed cube
  CGAL_assertion(Cube1 == Cube1.closure());
  CGAL_assertion(Cube1 == Cube1.regularization());
  CGAL_assertion((N1 - N1.boundary()) == N1.interior());
  CGAL_assertion(I1.closure() ==
   I1.complement().interior().complement());
  CGAL_assertion(I1.regularization() == I1.interior().closure());
  return 0;
}
```

File: examples/Nef_3/point_set_operations.cpp

29.5.3 Transformation

Using the *std::transform* function, a Nef polyhedron can be translated, rotated and scaled. The usage is shown in the following example:

```
#include <CGAL/Gmpz.h>
#include <CGAL/Extended_homogeneous.h>
#include <CGAL/Nef_polyhedron_3.h>
#include <CGAL/IO/Nef_polyhedron_iostream_3.h>


//instead of
//typedef CGAL::Extended_homogeneous<CGAL::Gmpz>  Kernel;
// workaround for VC++
struct Kernel : public CGAL::Extended_homogeneous<CGAL::Gmpz> {};

typedef CGAL::Nef_polyhedron_3<Kernel>  Nef_polyhedron;
typedef Nef_polyhedron::Plane_3  Plane_3;
typedef Nef_polyhedron::Vector_3  Vector_3;
typedef Nef_polyhedron::Aff_transformation_3  Aff_transformation_3;

int main() {

  Nef_polyhedron N(Plane_3(0,1,0,0));
  Aff_transformation_3 transl(CGAL::TRANSLATION, Vector_3(5, 7, 9));
  Aff_transformation_3 rotx90(1,0,0,
```

```
                            0,0,-1,
                            0,1,0,
                            1);
  Aff_transformation_3 scale(3,0,0,
                           0,3,0,
                           0,0,3,
                           2);

  N.transform(transl);
  CGAL_assertion(N == Nef_polyhedron(Plane_3(0,1,0,-7)));
  N.transform(rotx90);
  CGAL_assertion(N == Nef_polyhedron(Plane_3(0,0,1,-7)));
  N.transform(scale);
  CGAL_assertion(N == Nef_polyhedron(Plane_3(0,0,2,-21)));

  return 0;
}
```

File: examples/Nef_3/transformation.cpp

29.5.4 The Interface between `Polyhedron_3` and `Nef_polyhedron_3`

Nef_polyhedron_3 provides an interface for the conversion between polyhedral surfaces represented with the *CGAL::Polyhedron_3* class and *Nef_polyhedron_3*. *Polyhedron_3* represents orientable 2-manifold objects with boundaries. However, we exclude surfaces with boundaries from the conversion to *Nef_polyhedron_3* since they have no properly defined volume.

Both conversion directions can only be performed if the boundary of the point set is an oriented closed 2-manifold. *Nef_polyhedron_3* provides the function *is_simple()* and *Polyhedron_3* provides the function *is_closed()* to test for this property. The usage is illustrated by the example program below.

The conversion gives us the possibility to use several file formats. *Polyhedron_3* can read the (`.off`) file format and can write the (`.off`), OpenInventor (`.iv`), VRML 1.0 and 2.0 (`.wrl`) and Wavefront Advanced Visualizer object format (`.obj`), see Section 25.4.

```
#include <CGAL/Gmpz.h>
#include <CGAL/Homogeneous.h>
#include <CGAL/Polyhedron_3.h>
#include <CGAL/IO/Polyhedron_iostream.h>
#include <CGAL/Nef_polyhedron_3.h>
#include <CGAL/IO/Nef_polyhedron_iostream_3.h>
#include <iostream>

typedef CGAL::Homogeneous<CGAL::Gmpz>  Kernel;
typedef CGAL::Polyhedron_3<Kernel>  Polyhedron;
typedef CGAL::Nef_polyhedron_3<Kernel> Nef_polyhedron;
typedef Kernel::Vector_3  Vector_3;
typedef Kernel::Aff_transformation_3  Aff_transformation_3;

int main() {
  Polyhedron P;
  std::cin >> P;
```

```
  if(P.is_closed()) {
    Nef_polyhedron N1(P);
    Nef_polyhedron N2(N1);
    Aff_transformation_3 aff(CGAL::TRANSLATION, Vector_3(2,2,0,1));
    N2.transform(aff);
    N1 += N2;

    if(N1.is_simple()) {
      N1.convert_to_polyhedron(P);
      std::cout << P;
    }
    else
      std::cerr << "N1 is not a 2-manifold." << std::endl;
  }
}
```

File: examples/Nef_3/interface_polyhedron.cpp

29.5.5 Using an Extended Kernel

The provided extended kernels are used the same way as any other CGAL kernel. The essential difference is, that coordinates are not represented by the number type that was used to parameterize the kernel type, but by a *Nef_polynomial* parametrized by that number type.

The example iterates all vertices of a given Nef polyhedron and decides whether it is an standard vertex or a vertex on the infimaximal box. Furthermore, it tests whether any of the vertices is at (R,R,R). Recall that R was the symbolical value, large but finite, for the size of the infimaximal box.

```
#include <CGAL/Gmpz.h>
#include <CGAL/Extended_homogeneous.h>
#include <CGAL/Nef_polyhedron_3.h>
#include <CGAL/IO/Nef_polyhedron_iostream_3.h>

typedef CGAL::Gmpz  NT;
//instead of
//typedef CGAL::Extended_homogeneous<NT>  Kernel;
// workaround for VC++
struct Kernel : public CGAL::Extended_homogeneous<NT> {};

typedef CGAL::Nef_polyhedron_3<Kernel>  Nef_polyhedron;
typedef Nef_polyhedron::RT  RT;
typedef Nef_polyhedron::Point_3  Point_3;
typedef Nef_polyhedron::Plane_3  Plane_3;
typedef Nef_polyhedron::Vertex_const_iterator  Vertex_const_iterator;

int main() {

  Nef_polyhedron N;
  std::cin >> N;

  Vertex_const_iterator v;
  for(v = N.vertices_begin(); v != N.vertices_end(); ++v) {
```

```
    Point_3 p(v->point());
    if(p.hx().degree() > 0 || p.hy().degree() > 0 || p.hz().degree() > 0)
      std::cout << "extended vertex at " << p << std::endl;
    else
      std::cout << "standard vertex at " << p << std::endl;

    if(p == Point_3(RT(0,1), RT(0,1), RT(0,1)))
       std::cout << "  found vertex (right,back,top) of the infimaximal
   box"
                  << std::endl;
  }

  return 0;
}
```

File: examples/Nef_3/extended_kernel.cpp

29.6 File I/O

Nef_polyhedron_3 provides an input and an output operator for a proprietary file format. It includes the complete incidence structure, the geometric data, and the marks of each item. The output depends on the output operators of the geometric primitives provided by the traits class, and on the output operators of the used number type. Therefore, it is necessary to use the same kernel and the same number type for input and output operations.

We recommend the use of the CGAL kernels *Homogeneous*, *Exact_predicates_exact_constructions_kernel*, or *Extended_homogeneous*. The Homogeneous kernel provides reliable fast performance. In combination with *leda_integer* it is the fastest kernel for *Nef_polyhedron_3*. The *Exact_predicates_exact_constructions_kernel* uses filtering. In non-degenerate scenarios it's faster than the *Homogeneous* kernel. The most important advantage of the filtered kernel is that it is a Cartesian kernel, which allows the proper handling of OFF files using floating-point coordinates.

For effective filtering we had to change some concepts. The new concepts must be activated by using the *SNC_indexed_items*, because they don't apply for the extended kernels, yet. The new concepts also speed up *Nef_polyhedron_3* also in combination with all other standard kernels. The following example illustrates their usage.

```
#include <CGAL/Exact_predicates_exact_constructions_kernel.h>
#include <CGAL/Polyhedron_3.h>
#include <CGAL/IO/Polyhedron_iostream.h>
#include <CGAL/Nef_polyhedron_3.h>
#include <CGAL/IO/Nef_polyhedron_iostream_3.h>

typedef CGAL::Exact_predicates_exact_constructions_kernel Kernel;
typedef CGAL::Polyhedron_3<Kernel>  Polyhedron;
typedef CGAL::Nef_polyhedron_3<Kernel>  Nef_polyhedron;

int main() {

  Polyhedron P;
  std::cin >> P;
  Nef_polyhedron N(P);
```

```
  std::cout << "Exact_predicates_exact_constructions_kernel +
   SNC_indexed_items"
            << std::endl
            << "  allows efficient handling of input "
                 "using floating point coordinates"
            << std::endl;

  if(N.is_simple()) {
    N.convert_to_polyhedron(P);
    std::cout << P;
  }
  else {
    std::cout << N;
  }
}
```

File: examples/Nef_3/handling_double_coordinates.cpp

We provide compatibility between the input and output of various kernels. For most of the CGAL kernels it is possible to write a file constructed with one kernel and reread it with another. Also, it is possible to write a bounded Nef polyhedron using the *Extended_homogeneous* kernel and to read it afterwards using a standard kernel.

```
#include <CGAL/Gmpz.h>
#include <CGAL/Homogeneous.h>
#include <CGAL/Extended_homogeneous.h>
#include <CGAL/Nef_polyhedron_3.h>
#include <CGAL/IO/Nef_polyhedron_iostream_3.h>
#include <fstream>

typedef CGAL::Gmpz  NT;
typedef CGAL::Homogeneous<NT>  SK;
typedef CGAL::Extended_homogeneous<NT>  EK;
typedef CGAL::Nef_polyhedron_3<SK>  Nef_polyhedron_S;
typedef CGAL::Nef_polyhedron_3<EK>  Nef_polyhedron_E;

int main() {
  Nef_polyhedron_E E;
  Nef_polyhedron_S S;

  std::cin >> E;

  if(E.is_bounded()) {
    std::ofstream out("temp.nef3");
    out << E;
    std::ifstream in("temp.nef3");
    in >> S;
  }
}
```

File: examples/Nef_3/nefIO.cpp

29.7 Further Example Programs

29.7.1 Exploring a Sphere Map

A sphere map is explored by using the function *get_sphere_map*, which returns the sphere map of the specified vertex as a *Nef_polyhedron_S2*. *Nef_polyhedron_S2* provides the functionality necessary for the exploration. Note, that one has to use the type *Nef_polyhedron_S2* as specified in *Nef_polyhedron_3* as is shown in the following example.

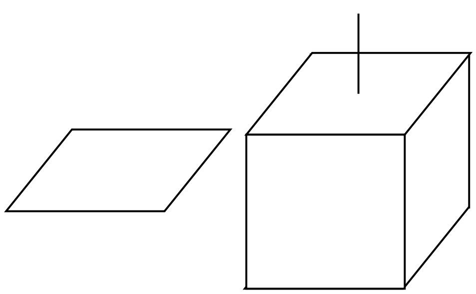
```
#include <CGAL/Gmpz.h>
#include <CGAL/Homogeneous.h>
#include <CGAL/Nef_polyhedron_3.h>
#include <CGAL/IO/Nef_polyhedron_iostream_3.h>

typedef CGAL::Homogeneous<CGAL::Gmpz> Kernel;
typedef CGAL::Nef_polyhedron_3<Kernel> Nef_polyhedron_3;


int main() {

  // We've put the typedefs here as VC7 gives us an ICE if they are
   global typedefs
  typedef Nef_polyhedron_3::Vertex_const_iterator Vertex_const_iterator;
  typedef Nef_polyhedron_3::Nef_polyhedron_S2 Nef_polyhedron_S2;
  typedef Nef_polyhedron_S2::SVertex_const_handle SVertex_const_handle;
  typedef Nef_polyhedron_S2::SHalfedge_const_handle
   SHalfedge_const_handle;
  typedef Nef_polyhedron_S2::SHalfloop_const_handle
   SHalfloop_const_handle;
  typedef Nef_polyhedron_S2::SFace_const_iterator SFace_const_iterator;
  typedef Nef_polyhedron_S2::SFace_cycle_const_iterator
    SFace_cycle_const_iterator;

  Nef_polyhedron_3 N;
  std::cin >> N;

  Vertex_const_iterator v = N.vertices_begin();
  Nef_polyhedron_S2 S(N.get_sphere_map(v));

  int i=0;
  SFace_const_iterator sf;
  for(sf = S.sfaces_begin(); sf != S.sfaces_end(); sf++) {
    SFace_cycle_const_iterator it;
    std::cout << "the sface cycles of sface " << i++ << " start with
   an\n";
    for(it = sf->sface_cycles_begin(); it != sf->sface_cycles_end();
   it++) {
      if (it.is_svertex())
        std::cout << "  svertex at position "
                  << SVertex_const_handle(it)->point() << std::endl;
      else if (it.is_shalfedge())
        std::cout << "  shalfedge from "
                  << SHalfedge_const_handle(it)->source()->point() << "
```

```
  to "
                  << SHalfedge_const_handle(it)->target()->point() <<
  std::endl;
     else if (it.is_shalfloop())
       std::cout << "  shalfloop lying in the plane "
                  << SHalfloop_const_handle(it)->circle() << std::endl;
     // other cases can not occur.
    }
  }

  return 0;
}
```

File: examples/Nef_3/exploration_SM.cpp

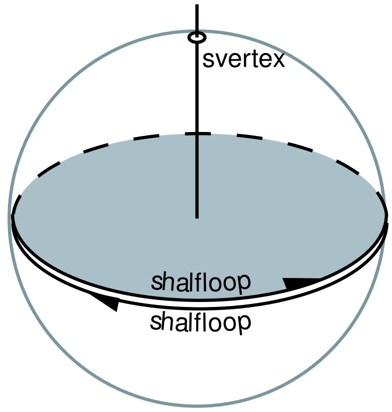
29.7.2 Exploring Shells

A *shell* of a Nef polyhedron is the connected part of the surface incident to a certain volume. Each halffacet, sface and shalfedge belongs to a single shell. The figure below illustrates the notion of a shell. It shows a Nef polyhedron with two volumes and three shells.



The first volume is the outer volume and the second volume is the interior of the cube. The first shell is the whole surface of the left object. The second shell is the outer surface of the right object, and the third shell is the inner surface of the right object.

In detail, the first shell consists of two halffacets, eight halfedges and four vertices. The second shell consists of the eight vertices of the cube plus the two endpoints of the antenna, all halffacets oriented outwards, and all halfedges. The third shell consists of the same eight vertices of the cube, plus the endpoint of the antenna that is in contact with the cube, all halffacets oriented inwards, and all halfedges (the same as for the second shell).



We discuss how sfaces, shalfedges, and sloops belong to the shells with a closeup view of the situation at the antenna foot. As you can see, there are three items on the sphere map - a shalfloop for each halffacet which intersects the sphere, and an svertex where the antenna intersects the sphere. The upper shalfloop lies on the halffacet which is oriented outwards and is therefore also oriented outwards. This shalfloop and the svertex belong to the second shell. The other shalfloop lies on the inwards oriented halffacet and is oriented inwards, too. This shalfloop belongs to the third shell.

Nef_polyhedron_3 offers a visitor interface to explore a shell following the well-known visitor pattern [GHJV95]. The interface is illustrated by the following example.

```
#include <CGAL/Gmpz.h>
#include <CGAL/Homogeneous.h>
#include <CGAL/Nef_polyhedron_3.h>
#include <CGAL/IO/Nef_polyhedron_iostream_3.h>

typedef CGAL::Homogeneous<CGAL::Gmpz> Kernel;
typedef CGAL::Nef_polyhedron_3<Kernel> Nef_polyhedron;
typedef Nef_polyhedron::Vertex_const_handle Vertex_const_handle;
typedef Nef_polyhedron::Halfedge_const_handle Halfedge_const_handle;
typedef Nef_polyhedron::Halffacet_const_handle Halffacet_const_handle;
typedef Nef_polyhedron::SHalfedge_const_handle SHalfedge_const_handle;
typedef Nef_polyhedron::SHalfloop_const_handle SHalfloop_const_handle;
typedef Nef_polyhedron::SFace_const_handle SFace_const_handle;
typedef Nef_polyhedron::Volume_const_iterator Volume_const_iterator;
typedef Nef_polyhedron::Shell_entry_const_iterator
   Shell_entry_const_iterator;
typedef Kernel::Point_3 Point_3;

class Shell_explorer {
  bool first;
  const Nef_polyhedron& N;
  Vertex_const_handle v_min;

public:
  Shell_explorer(const Nef_polyhedron& N_)
    : first(true), N(N_) {}

  void visit(Vertex_const_handle v) {
    if(first ||
   CGAL::lexicographically_xyz_smaller(v->point(),v_min->point())) {
      v_min = v;
```

```
      first=false;
    }
  }

  void visit(Halfedge_const_handle ) {}
  void visit(Halffacet_const_handle ) {}
  void visit(SHalfedge_const_handle ) {}
  void visit(SHalfloop_const_handle ) {}
  void visit(SFace_const_handle ) {}

  Vertex_const_handle& minimal_vertex() { return v_min; }
  void reset_minimal_vertex() { first = true; }
};

int main() {
  Nef_polyhedron N;
  std::cin >> N;

  int ic = 0;
  Volume_const_iterator c;
  Shell_explorer SE(N);
  CGAL_forall_volumes(c,N) {
    std::cout << "Volume " << ic++ << std::endl;
    int is = 0;
    Shell_entry_const_iterator it;
    CGAL_forall_shells_of(it,c) {
      SE.reset_minimal_vertex();
      N.visit_shell_objects(SFace_const_handle(it),SE);
      Point_3 p(SE.minimal_vertex()->point());
      std::cout << "  minimal vertex of shell " << is++
                << " is at " << p << std::endl;
    }
  }
}
```

File: examples/Nef_3/shell_exploration.cpp

The function *visit_shell_objects(SFace_const_handle sf, Visitor& V)* explores a shell starting at the *sf*. The second argument expects any class providing the (possibly empty) functions *visit(Vertex_const_handle)*, *visit(Halfedge_const_handle)* (remember that Halfedge is the same type as SVertex), *visit(Halffacet_const_handle)*, *visit(SHalfedge_const_handle)*, *visit(SHalfloop_const_handle)* and *visit(SFace_const_handle)*. The *visit_shell_objects* function will call *visit* for each item belonging to the shell once. There are no further requirements on that class.

In the example, the class *Shell_explorer* is passed as second argument to *visit_shell_objects*. Its task is to find the lexicographically smallest vertex of a shell. Its internal state consists of three variables. The first one is a reference to the explored Nef polyhedron. This reference is often necessary to retrieve information from the Nef polyhedron. The second variable *v_min* stores the smallest vertex found so far, and the third variable *first* is initialized to *false* to signal that no vertex has been visited so far. After the first vertex has been visited *first* is changed to *true*.

Shell_explorer provides further member functions. After the exploration of a shell the *minimal_vertex* function retrieves the smallest vertex. The *reset_minimal_vertex* function allows one to use the same instance of

Shell_explorer on multiple shells. In this case, the *reset_minimal_vertex* function has to be called between the exploration of two shells.

The example program uses the *Shell_explorer* for each shell of the given Nef polyhedron once and reports the smallest vertex of each shell to the standard output.

29.7.3 Point Location

The *locate(Point_3 p)* function locates the point *p* in the Nef polyhedron and returns the item the point belongs to. The *locate* function returns an instance of *Object_handle*, which is a polymorphic handle type representing any handle type, no matter if it is mutable or const. For further usage of the result, the *Object_handle* has to be casted to the concrete handle type. The *CGAL::assign* function performs such a cast. It returns a boolean that reports the success or the failure of of the cast. Looking at the possible return values of the *locate* function, the *Object_handle* can represent a *Vertex_const_handle*, a *Halfedge_const_handle*, a *Halffacet_handle*, or a *Volume_const_handle*. One of the four casts will succeed.

```
#include <CGAL/Gmpz.h>
#include <CGAL/Homogeneous.h>
#include <CGAL/Nef_polyhedron_3.h>
#include <CGAL/IO/Nef_polyhedron_iostream_3.h>

typedef CGAL::Homogeneous<CGAL::Gmpz> Kernel;
typedef CGAL::Nef_polyhedron_3<Kernel> Nef_polyhedron_3;
  typedef Kernel::Point_3 Point_3;

int main() {
  //We've put the typedefs here as VC7 gives us an ICE if they are global
     typedefs
  typedef Nef_polyhedron_3::Vertex_const_handle Vertex_const_handle;
  typedef Nef_polyhedron_3::Halfedge_const_handle Halfedge_const_handle;
  typedef Nef_polyhedron_3::Halffacet_const_handle
   Halffacet_const_handle;
  typedef Nef_polyhedron_3::Volume_const_handle Volume_const_handle;
  typedef Nef_polyhedron_3::Object_handle Object_handle;


  Nef_polyhedron_3 N;
  std::cin >> N;

  Vertex_const_handle v;
  Halfedge_const_handle e;
  Halffacet_const_handle f;
  Volume_const_handle c;
  Object_handle o = N.locate(Point_3(0,0,0));
  if(CGAL::assign(v,o))
    std::cout << "Locating vertex" << std::endl;
  else if(CGAL::assign(e,o))
    std::cout << "Locating edge" << std::endl;
  else if(CGAL::assign(f,o))
    std::cout << "Locating facet" << std::endl;
  else if(CGAL::assign(c,o))
    std::cout << "Locating volume" << std::endl;
  //other cases can not occur
```

```
  return 0;
}
```

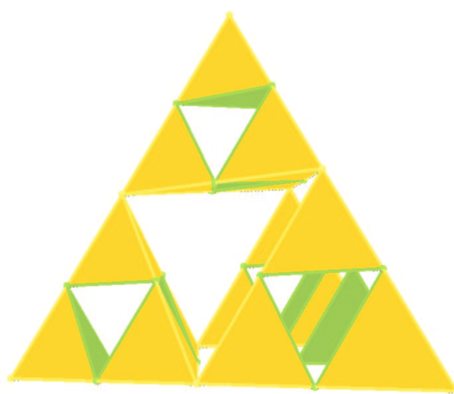
File: examples/Nef_3/nef_3_point_location.cpp

29.8 Visualization

With the *Qt_widget_OpenGL* class an interface to OpenGL visualization via Qt is offered. The class knows how to handle mouse movements and clicks and how to move and scale the 3D object displayed in the widget. *Qt_widget_OpenGL* is a basis for writing Qt widgets displaying 3D objects. A user can derive a new class from *Qt_widget_OpenGL* which implements the drawing method and configures the context menus.

29.8.1 Visualizing a 3D Nef polyhedron

Qt_widget_Nef_3 implements the drawing methods for displaying instances of *Nef_polyhedron_3*. The following example shows how to set up an QApplication with a main widget of type *Qt_widget_Nef_3* and how to start the viewer.



```
#include <CGAL/basic.h>
#include <CGAL/Gmpz.h>
#include <CGAL/Homogeneous.h>
#include <CGAL/Nef_polyhedron_3.h>
#include <CGAL/IO/Nef_polyhedron_iostream_3.h>
#include <CGAL/IO/Qt_widget_Nef_3.h>
#include <qapplication.h>

typedef CGAL::Homogeneous<CGAL::Gmpz> Kernel;
typedef CGAL::Nef_polyhedron_3<Kernel> Nef_polyhedron_3;

int main(int argc, char* argv[]) {
  Nef_polyhedron_3 N;
  std::cin >> N;
```

```
  QApplication a(argc, argv);
  CGAL::Qt_widget_Nef_3<Nef_polyhedron_3>* w =
    new CGAL::Qt_widget_Nef_3<Nef_polyhedron_3>(N);
  a.setMainWidget(w);
  w->show();
  return a.exec();
}
```

File: demo/Nef_3/visualization_SNC.cpp

29.8.2 Visualizing a Sphere Map

Qt_widget_Nef_S2 is a widget implemented on the basis of *Qt_widget_OpenGL*. It can be used to visualize the sphere map of a vertex in a *Nef_polyhedron_3* using the interface between *Nef_polyhedron_S2* and *Nef_polyhedron_3*.

```
#include <CGAL/basic.h>
#include <CGAL/Gmpz.h>
#include <CGAL/Homogeneous.h>
#include <CGAL/Nef_polyhedron_3.h>
#include <CGAL/IO/Nef_polyhedron_iostream_3.h>
#include <CGAL/IO/Qt_widget_Nef_S2.h>
#include <qapplication.h>

typedef CGAL::Homogeneous<CGAL::Gmpz> Kernel;
typedef CGAL::Nef_polyhedron_3<Kernel> Nef_polyhedron_3;



int main(int argc, char* argv[]) {

  // We've put the typedefs here as VC7 gives us an ICE if they are
   global typedefs
  typedef Nef_polyhedron_3::Vertex_const_iterator Vertex_const_iterator;
  typedef Nef_polyhedron_3::Nef_polyhedron_S2 Nef_polyhedron_S2;

  Nef_polyhedron_3 N;
  std::cin >> N;
  Vertex_const_iterator v = N.vertices_begin();
  Nef_polyhedron_S2 S(N.get_sphere_map(v));

  QApplication a(argc, argv);
  CGAL::Qt_widget_Nef_S2<Nef_polyhedron_S2>* w =
    new CGAL::Qt_widget_Nef_S2<Nef_polyhedron_S2>(S);
  a.setMainWidget(w);
  w->show();
  return a.exec();
}
```

File: demo/Nef_3/visualization_SM.cpp

3D Boolean Operations on Nef Polyhedra Reference Manual

Peter Hachenberger, Lutz Kettner, and Michael Seel

A Nef polyhedron is any point set generated from a finite number of open halfspaces by set complement and set intersection operations. In our implementation of Nef polyhedra in 3-dimensional space, we offer a B-rep data structures that is closed under boolean operations and with all their generality. Starting from halfspaces (and also directly from oriented 2-manifolds), we can work with set union, set intersection, set difference, set complement, interior, exterior, boundary, closure, and regularization operations. In essence, we can evaluate a CSG-tree with halfspaces as primitives and convert it into a B-rep representation.

In fact, we work with two data structures; one that represents the local neighborhoods of vertices, which is in itself already a complete description, and a data structure that connects these neighborhoods up to a global data structure with edges, facets, and volumes. We offer a rich interface to investigate these data structures, their different elements and their connectivity. We provide affine (rigid) tranformations and a point location query operation. We have a custom file format for storing and reading Nef polyhedra from files. We offer a simple OpenGL visualization for debugging and illustrations.

29.9 Classified Reference Pages

Classes

Functions

template<class Nef_polyhedron_3>

std::size_t

OFF_to_nef_3(std::istream& in, Nef_polyhedron_3& N)

page 2035

template <class Traits>
ostream& *ostream& out << CGAL::Nef_polyhedron_3<Traits> N*

page 2037

template <class Traits>
istream& *istream& in >> CGAL::Nef_polyhedron_3<Traits>& N*

page 2038

29.10 Alphabetical List of Reference Pages

CGAL::Nef_polyhedron_3<Traits>

Definition

A 3D Nef polyhedron is a subset of the 3-dimensional space that is the result of forming complements and intersections starting from a finite set H of 3-dimensional halfspaces. Nef polyhedra are closed under all binary set operations, i.e., intersection, union, difference, complement, and under the topological operations boundary, closure, and interior.

A 3D Nef polyhedron can be represented by the local pyramids of the minimal elements of its incidence structure. Without going into to much detail, a local pyramid essentially reflects the topologic and geometric situation at a certain location in a point set. For finite polyhedra the minimal elements of the incidence structure are vertices only. This means, that it suffices to model the topological and geometric situation of the vertices. For 3D Nef polyhedra, the local pyramid of a vertex is represented by a planar Nef polyhedra embedded on a sphere.

A *Nef_polyhedron_3* consists of vertices V, a sphere map for each vertex in V, edges E, facets F, volumes C, a mark for every item, and an incidence relation on them. Each edge and each facet is represented by two halfedges or two halffacets, respectively.

#include <CGAL/Nef_polyhedron_3.h>

Parameters

```
template < class Nef_polyhedronTraits_3,
           class Nef_polyhedronItems_3 = CGAL::Default_items<Nef_polyhedronTraits_3>,
           class Nef_polyhedronMarks = bool >
class Nef_polyhedron_3;
```

The first parameter requires one of the following exact kernels: *Homogeneous*, *Simple_homogeneous*, *Extended_homogeneous* parametrized with *Gmpz*, *leda_integer* or any other number type modeling $\mathbb{Z}$, or *Cartesian*, *Simple_cartesian*, *Extended_cartesian* parametrized with *Gmpq*, *leda_rational*, *Quotient<Gmpz>* or any other number type modeling $\mathbb{Q}$.

The second parameter and the third parameter are for future considerations. Neither *Nef_polyhedronItems_3* nor *Nef_polyhedronMarks* is specifed, yet. Do not use any other than the default types for these two template parameters.

Types

Nef_polyhedron_3<Traits>:: Traits	traits class selected for *Nef_polyhedronTraits_3*.
Nef_polyhedron_3<Traits>:: Mark	All object (vertices, edges, etc.) are attributed by a Mark. Mark equals bool.
Nef_polyhedron_3<Traits>:: size_type	size type of *Nef_polyhedron_3*.
Nef_polyhedron_3<Traits>:: Vertex_const_handle	non-mutable handle to a vertex.
Nef_polyhedron_3<Traits>:: Halfedge_const_handle	non-mutable handle to a halfedge.
Nef_polyhedron_3<Traits>:: Halffacet_const_handle	non-mutable handle to a halffacet.

Nef_polyhedron_3<Traits>:: Volume_const_handle non-mutable handle to a volume.
Nef_polyhedron_3<Traits>:: SVertex_const_handle non-mutable handle to a svertex.
Nef_polyhedron_3<Traits>:: SHalfedge_const_handle non-mutable handle to a shalfedge.
Nef_polyhedron_3<Traits>:: SHalfloop_const_handle non-mutable handle to a shalfloop.
Nef_polyhedron_3<Traits>:: SFace_const_handle non-mutable handle to a sface.

Nef_polyhedron_3<Traits>:: Vertex_const_iterator non-mutable iterator over all vertices.
Nef_polyhedron_3<Traits>:: Halfedge_const_iterator non-mutable iterator over all halfeges.
Nef_polyhedron_3<Traits>:: Halffacet_const_iterator non-mutable iterator over all halffacets.
Nef_polyhedron_3<Traits>:: Volume_const_iterator non-mutable iterator over all volumes.
Nef_polyhedron_3<Traits>:: SVertex_const_iterator non-mutable iterator over all svertices.
Nef_polyhedron_3<Traits>:: SHalfedge_const_iterator non-mutable iterator over all shalfedges.
Nef_polyhedron_3<Traits>:: SHalfloop_const_iterator non-mutable iterator over all shalfloops.
Nef_polyhedron_3<Traits>:: SFace_const_iterator non-mutable iterator over all sfaces.

Nef_polyhedron_3<Traits>:: SHalfedge_around_svertex_const_circulator

non-mutable circulator of shalfedges around a svertex (cw).

Nef_polyhedron_3<Traits>:: SHalfedge_around_sface_const_circulator

non-mutable circulator of shalfedges around a sface (ccw).

Nef_polyhedron_3<Traits>:: SHalfedge_around_facet_const_circulator

non-mutable circulator of shalfedges around a halffacet (ccw).

Nef_polyhedron_3<Traits>:: SFace_cycle_const_iterator

non-mutable iterator over the cylces of a sface.

Nef_polyhedron_3<Traits>:: Halffacet_cycle_const_iterator

non-mutable iterator over the cylces of a halffacet.

Nef_polyhedron_3<Traits>:: Shell_entry_const_iterator

non-mutable iterator providing an entry to each shell.

Nef_polyhedron_3<Traits>:: Object_handle a generic handle to an object. The kind of object *(vertex, halfedge, halffacet, volume, svertex, shalfedge, shalfloop, sface)* can be determined and the object can be assigned to a corresponding constant handle by one of the following functions:
bool assign(Vertex_const_handle& h, Object_handle)
bool assign(Halfedge_const_handle& h, Object_handle)
bool assign(Halffacet_const_handle& h, Object_handle)
bool assign(Volume_const_handle& h, Object_handle)
bool assign(SVertex_const_handle& h, Object_handle)
bool assign(SHalfedge_const_handle& h, Object_handle)
bool assign(SHalfloop_const_handle& h, Object_handle)
bool assign(SFace_const_handle& h, Object_handle)
where each function returns *true* iff the assignment to *h* could be accomplished.

Nef_polyhedron_3<Traits>:: Point_3 location of vertices.

Nef_polyhedron_3<Traits>:: Segment_3 segment represented by a halfedge.
Nef_polyhedron_3<Traits>:: Vector_3 direction of a halfedge.
Nef_polyhedron_3<Traits>:: Plane_3 plane of a halffacet lies in.
Nef_polyhedron_3<Traits>:: Aff_transformation_3

affine transformation.

Nef_polyhedron_3<Traits>:: Polylines_tag tag for calling polyline constructor.

enum Boundary { EXCLUDED, INCLUDED};

construction selection.

enum Content { EMPTY, COMPLETE}; construction selection.
enum Intersection_mode { CLOSED_HALFSPACE, OPEN_HALFSPACE, PLANE_ONLY};

intersection selection.

Nef_polyhedron_3<Traits>:: Nef_polyhedron_S2

a sphere map.

Nef_polyhedron_3<Traits>:: Polyhedron a polyhedral surface.

Creation

Nef_polyhedron_3<Traits> N(Content space = EMPTY);

creates a Nef polyhedron and initializes it to the empty set if *plane == EMPTY* and to the whole space if *space == COMPLETE*.

Nef_polyhedron_3<Traits> N(Plane_3 p, Boundary b = INCLUDED);

creates a Nef polyhedron containing the halfspace on the negative side of *p* including *p* if *b==INCLUDED*, excluding *p* if *b==EXCLUDED*.

Nef_polyhedron_3<Traits> N(Polyhedron& P);

creates a Nef polyhedron, which represents the same point set as the polyhedral surface *P* does.

Nef_polyhedron_3<Traits> N(Input_iterator begin, Input_iterator end);

creates a Nef polyhedron consisting of a single polygon spanned by the list of points in the iterator range *[begin,end)*. If the points do not on a common supporting plane, the constructor tries to triangulate the polygon into multiple facets.If the construction does not succeed, the empty set is created.

template <class Forward_iterator>
Nef_polyhedron_3<Traits> N(Forward_iterator it, Forward_iterator end, Polylines_tag);

The iterator range [it, end) defines a set point ranges, each of which defines a polyline.

Access Member Functions

bool	*N.is_simple() const*	returns true, if *N* is a 2-manifold.
bool	*N.is_valid() const*	checks the integrity of *N* .
Size_type	*N.number_of_vertices() const*	returns the number of vertices.
Size_type	*N.number_of_halfedges() const*	return the number of halfedges.
Size_type	*N.number_of_edges() const*	returns the number of halfedge pairs.
Size_type	*N.number_of_halffacets() const*	returns the number of halffacets.
Size_type	*N.number_of_facets() const*	returns the number of halffacet pairs.
Size_type	*N.number_of_volumes() const*	returns the number of volumes.
Vertex_const_iterator	*N.vertices_begin() const*	iterator over all vertices.
Vertex_const_iterator	*N.vertices_end() const*	past-the-end iterator.
Halfedge_const_iterator	*N.halfedges_begin() const*	iterator over all halfedges.
Halfedge_const_iterator	*N.halfedges_end() const*	past-the-end iterator.
Halffacet_const_iterator	*N.halffacets_begin() const*	iterator over all halffacets.
Halffacet_const_iterator	*N.halffacets_end() const*	past-the-end iterator.
Volume_const_iterator	*N.volumes_begin() const*	iterator over all volumes.
Volume_const_iterator	*N.volumes_end() const*	past-the-end iterator.

The following macros are provided: *CGAL_forall_vertices(v,N)*, *CGAL_forall_halfedges(e,N)*, *CGAL_forall_edges(e,N)*, *CGAL_forall_halffacets(f,N)*, *CGAL_forall_facets(f,N)*, *CGAL_forall_volumes(c,N)* where *N* is a *Nef_polyhedron_3*.

Object_handle	*N.locate(Point_3 p) const*	
		returns a generic handle to the object (vertex, edge, facet or volume) which contains the point p in its relative interior.
Nef_polyhedron_S2	*N.get_sphere_map(Vertex_const_iterator v) const*	
		returns the neighborhood of a vertex modeled by a *Nef_polyhedron_S2*.

Point Set Predicates

bool	*N.is_empty() const*	returns true, if *N* is the empty point set.
bool	*N.is_space() const*	returns true, if *N* is the complete 3D space.
bool	$N == N1$	returns true, if *N* and N1 comprise the same point sets.
bool	$N\ != N1$	returns true, if *N* and N1 comprise different point sets.
bool	$N < N1$	returns true, if *N* is a proper subset of N1.
bool	$N > N1$	returns true, if *N* is a proper superset of N1.
bool	$N <= N1$	returns true, if *N* is a subset of N1.
bool	$N >= N1$	returns true, if *N* is a superset of N1.

Unary Set Operations

Nef_polyhedron_3<Traits>	*N.complement() const*	
		returns the complement of *N* .
Nef_polyhedron_3<Traits>	*N.interior() const*	returns the interior of *N* .
Nef_polyhedron_3<Traits>	*N.boundary() const*	
		returns the boundary of *N* .
Nef_polyhedron_3<Traits>	*N.closure() const*	returns the closure of *N* .
Nef_polyhedron_3<Traits>	*N.regularization() const*	
		returns the regularization, i.e. the closure of the interior, of *N* .
Nef_polyhedron_3<Traits>	*!N*	returns the complement of *N* .

Binary Set Operations

Nef_polyhedron_3<Traits>	*N.intersection(N1) const*	
		return the intersection of *N* and N1.
Nef_polyhedron_3<Traits>	*N.join(N1) const*	return the union of *N* and N1. (Note that "union" is a C++ keyword and cannot be used for this operation.)

Nef_polyhedron_3<Traits>	*N.difference(N1) const*	
		return the difference between N and N1.
Nef_polyhedron_3<Traits>	*N.symmetric_difference(N1) const*	
		return the symmetric difference of N and N1.
Nef_polyhedron_3<Traits>	*N.intersection(Plane_3 p, Intersection_mode im) const*	
		returns intersection of N with plane (*im=PLANE_ONLY*), open halfspace (*im=OPEN_HALFSPACE*), or closed halfspace (*im=CLOSED_HALFSPACE*). In the latter two cases, the halfspaces are on the negative side of the plane p. The function is written for the use with standard kernels, where halfspaces are not part of the domain. The function does not work in combination with an extended kernels or with an unbounded polyhedron.

Nef_polyhedron_3<Traits>	$N * N1$	return the intersection of N and N1.
Nef_polyhedron_3<Traits>	$N + N1$	return the union of N and N1.
Nef_polyhedron_3<Traits>	$N - N1$	return the difference between N and N1.
Nef_polyhedron_3<Traits>	$N\,\hat{}\,N1$	return the symmetric difference of N and N1.

void	$N *= N1$	intersects N and N1.
void	$N += N1$	unites N with N1.
void	$N -= N1$	subtracts N1 from N .
void	$N\,\hat{}= N1$	performs a symmetric intersection of N and N1.

Operations

void *N.clear(Content space = EMPTY)*

make N the empty set if *space == EMPTY* and the complete 3D space if *space == COMPLETE*.

void *N.transform(Aff_transformation_3 aff)*

applies an affine transformation to N .

void *N.convert_to_polyhedron(Polyhedron& P) const*

converts N into a Polyhedron.
Precondition: N is simple.

void *N.visit_shell_objects(SFace_const_handle f, Visitor& V) const*

calls the visit function of V for every item which belongs to the same shell as sf.

See Also

CGAL::Nef_polyhedron_3<Traits>::Vertex . . . . page 2020
CGAL::Nef_polyhedron_3<Traits>::Halfedge . . . . page 2021
CGAL::Nef_polyhedron_3<Traits>::Halffacet . . . . page 2023
CGAL::Nef_polyhedron_3<Traits>::Volume . . . . page 2025
CGAL::Nef_polyhedron_3<Traits>::SHalfedge . . . . page 2026
CGAL::Nef_polyhedron_3<Traits>::SHalfloop . . . . page 2029
CGAL::Nef_polyhedron_3<Traits>::SFace . . . . page 2031
CGAL::Nef_polyhedron_S2<Traits> . . . . page 1603
CGAL::Polyhedron_3<Traits> . . . . page 1795

Class

CGAL::Nef_polyhedron_3<Traits>::Vertex

Definition

A vertex is a point in the 3-dimensional space. Its incidence structure can be accessed creating a sphere map of the vertex. This is done by the member function *get_sphere_map* of the class *Nef_polyhedron_3*.

#include <CGAL/Nef_polyhedron_3.h>

Types

The following types are the same as in *Nef_polyhedron_3<Traits>*.

Vertex:: Mark	type of mark.
Vertex:: Point_3	point type stored in Vertex.

Creation

There is no need for a user to create a *Vertex* explicitly. The class *Nef_polyhedron_3<Traits>* manages the needed vertices internally.

Operations

Mark	*v.mark() const*	the mark of *v* .
Point_3	*v.point() const*	the point of *v* .

CGAL::Nef_polyhedron_3<Traits> . . . . page 2013
CGAL::Nef_polyhedron_S2<Traits> . . . . page 1603

CGAL::Nef_polyhedron_3<Traits>::Halfedge

Class

Definition

A Halfedge has a double meaning. In the global incidence structure of a *Nef_polyhedron_3* it is an oriented edge going from one vertex to another. A halfedge also coincides with an svertex of the sphere map of its source vertex. Because of this, we offer the types *Halfedge* and *SVertex* which are the same. Furthermore, the redundant functions *center_vertex()* and *source()* are provided. The reason is, that we get the same vertex either if we want to have the source vertex of a halfedge, or if we want to have the vertex in the center of the sphere map a svertex lies on. Figures 29.9 and 29.9 illustrate the incidence of a svertex on a sphere map and of a halfedge in the global structure.

As part of the global incidence structure, the member fuctions *source* and *target* return the source and target vertex of an edge. The member function *twin()* returns the opposite halfedge.

Looking at the incidence structure on a sphere map, the member function *out_sedge* returns the first outgoing shalfedge, and *incident_sface* returns the incident sface.

#include <CGAL/Nef_polyhedron_3.h>

Types

The following types are the same as in *Nef_polyhedron_3<Traits>*.

Halfedge:: Mark	type of mark.
Halfedge:: Sphere_point	sphere point type stored in Halfedge.
Halfedge:: Vertex_const_handle	const handle to vertex.
Halfedge:: Halfedge_const_handle	const handle to halfedge.
Halfedge:: SHalfedge_const_handle	const handle to SHalfedge.
Halfedge:: SFace_const_handle	const handle to SFace.

Creation

There is no need for a user to create a *Halfedge* explicitly. The class *Nef_polyhedron_3<Traits>* manages the needed halfedges internally.

Operations

Mark	*e.mark() const*	the mark of *e* .
Sphere_point	*e.point() const*	the sphere point of *e* .
bool	*e.is_isolated() const*	returns —true— if *e* has no adjacent sedges.

Vertex_const_handle	*e.center_vertex() const*	
		the center vertex of the sphere map *e* belongs to.
Vertex_const_handle	*e.source() const*	the source vertex of *e* .
Vertex_const_handle	*e.target() const*	the target vertex *e*.
Halfedge_const_handle	*e.twin() const*	the twin of *e* .
SHalfedge_const_handle		
	e.out_sedge() const	the first out sedge of *e* .
SFace_const_handle	*e.incident_sface() const*	
		the incident sface of *e* .

See Also

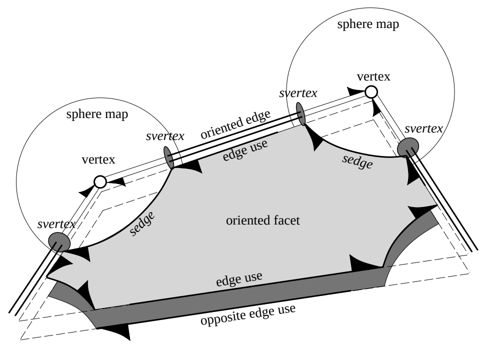
CGAL::Nef_polyhedron_3<Traits>::Vertex . . . . page 2020
CGAL::Nef_polyhedron_3<Traits>::SHalfedge . . . . page 2026
CGAL::Nef_polyhedron_3<Traits>::SFace . . . . page 2031
CGAL::Nef_polyhedron_S2<Traits>::Sphere_point . . . . page 1608

CGAL::Nef_polyhedron_3<Traits>::Halffacet

Class

Definition

A halffacet is an oriented, rectilinear bounded part of a plane. The following figure depicts the incidences to halfedges, vertices and the notion of facet cycles.



The member function *twin* returns the opposite halffacet, *incident_volume* returns the incident volume. A Halffacet cycle either consists of consecutive shalfedges along the border (or a hole) of the halffacet, or of a single shalfloop on the sphere map of a vertex isolated on the halffacet. The iterator range (*halffacet_cycles_begin()*/*halffacet_cycles_end()*) provides an entry element for each halffacet cycle of a halffacet.

#include <CGAL/Nef_polyhedron_3.h>

Types

The following types are the same as in *Nef_polyhedron_3<Traits>*.

Halffacet:: Mark	type of mark.
Halffacet:: Plane_3	plane type stored in Halffacet.
Halffacet:: Object_list	list of Object handles.
Halffacet:: Halffacet_const_handle	const handle to Halffacet.
Halffacet:: Volume_const_handle	const handle to volume.
Halffacet:: Halffacet_cycle_const_iterator	const iterator over the entries to all halffacet cycles of a halffacet.

Creation

There is no need for a user to create a *Halffacet* explicitly. The class *Nef_polyhedron_3<Traits>* manages the needed halffacets internally.

Operations

Mark	*f.mark() const*	the mark of f .
Plane_3	*f.plane() const*	the supporting plane of f .
Halffacet_const_handle	*f.twin() const*	the twin of f .
Volume_const_handle	*f.incident_volume() const*	the incident volume of f .
Halffacet_cycle_const_iterator	*f.facet_cycles_begin() const*	iterator over the entries to all halffacet cycles of f .
Halffacet_cycle_const_iterator	*f.facet_cycles_end() const*	past-the-end iterator.

See Also

CGAL::Nef_polyhedron_3<Traits>::Volume . . . . page 2025
CGAL::Nef_polyhedron_3<Traits>::Halfedge . . . . page 2021
CGAL::Nef_polyhedron_3<Traits>::SHalfedge . . . . page 2026

CGAL::Nef_polyhedron_3<Traits>::Volume

Class

Definition

A volume is a full-dimensional connected point set in $\mathbb{R}^3$. It is bounded by several shells, i.e. a connected part of the boundary incident to a single volume. An entry element to each shell is provided by the iterator range (*shells_begin()*/*shells_end()*). A *Shell_entry_iterator* is assignable to *SFace_handle*.

#include <CGAL/Nef_polyhedron_3.h>

Types

The following types are the same as in *Nef_polyhedron_3<Traits>*.

Volume:: Mark	type of mark.
Volume:: Object_list	list of Object handles.
Volume:: Volume_const_handle	const handle to Volume.
Volume:: Shell_entry_const_iterator	const iterator over the entries to all shells adjacent to a volume.

Creation

There is no need for a user to create a *Volume* explicitly. The class *Nef_polyhedron_3<Traits>* manages the needed volumes internally.

Operations

Mark	*c.mark() const*	the mark of *c* .
Shell_entry_const_iterator	*c.shells_begin() const*	const iterator over the entries to all shells adjacent to *c* .
Shell_entry_const_iterator	*c.shells_end() const*	past-the-end iterator.

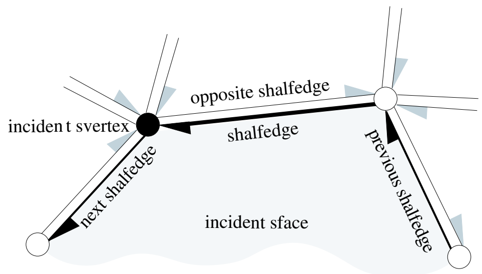
See Also

CGAL::Nef_polyhedron_3<Traits>::SFace . . . . page 2031

CGAL::Nef_polyhedron_3<Traits>::SHalfedge

Definition

A shalfedge is a great arc on a sphere map. Figure 29.9 depicts the relationship between a shalfedge and its incident shalfedges, svertices, and sfaces on a sphere map. A shalfedge is an oriented sedge between two svertices. It is always paired with a shalfedge pointing in the opposite direction. The *twin()* member function returns this shalfedge of opposite orientation.



The *snext()* member function points to the successor shalfedge around this sface while the *sprev()* member function points to the preceding shalfedge. An successive assignments of the form *se* = *se->snext()* cycles counterclockwise around the sface (or hole).

Similarly, the successive assignments of the form *se* = *se->snext()->twin()* cycle clockwise around the svertex and traverse all halfedges incident to this svertex. The assignment *se* = *se->cyclic_adj_succ()* can be used as a shortcut.

The role of shalfedges in a facet is illustrated in Figure 29.9. The *facet()* member function returns the facet in which the shalfedge is part of one of the facet cycles. The successive assignment of the form *se* = *se->next()* cycles counterclockwise around the facet (or a hole of the facet).

A const circulators is provided for each of the three circular orders. The circulators are bidirectional and assignable to *SHalfedge_const_handle*.

#include <CGAL/Nef_polyhedron_3.h>

Types

The following types are the same as in *Nef_polyhedron_3<Traits>*.

SHalfedge:: Mark	type of mark.
SHalfedge:: Sphere_circle	sphere circle type stored in SHalfedge.
SHalfedge:: Halffacet_const_handle	const handle to Halffacet.
SHalfedge:: SVertex_const_handle	const handle to SVertex.
SHalfedge:: SHalfedge_const_handle	const handle to SHalfedge.
SHalfedge:: SFace_const_handle	const handle to SFace.

Creation

There is no need for a user to create a *SHalfedge* explicitly. The class *Nef_polyhedron_3<Traits>* manages the needed shalfedges internally.

Operations

Mark	*se.mark() const*	the mark of *se* .
Sphere_circle	*se.circle() const*	the sphere circle of *se* .
SHalfedge_const_handle	*se.twin() const*	the twin of *se* .
SVertex_const_handle	*se.source() const*	the source svertex of *se* .
SVertex_const_handle	*se.target() const*	equals *twin()->source()*.
SHalfedge_const_handle	*se.prev() const*	the SHalfedge previous to *se* in a facet cycle.
SHalfedge_const_handle	*se.next() const*	the next SHalfedge of *se* in a facet cycle.
SHalfedge_const_handle	*se.sprev() const*	the SHalfedge previous to *se* in a sface cycle.
SHalfedge_const_handle	*se.snext() const*	the next SHalfedge of *se* in a sface cycle.
SHalfedge_const_handle	*se.cyclic_adj_pred() const*	the edge before *se* in the cyclic ordered adjacency list of source().
SHalfedge_const_handle	*se.cyclic_adj_succ() const*	the edge after *se* in the cyclic ordered adjacency list of source().
Halffacet_const_handle	*se.facet() const*	the facet that corresponds to *se* in the 3D incidence structure.
SFace_const_handle	*se.incident_sface() const*	the incident sface of *se* .
bool	*se.in_outer_sface_cycle() const*	determines whether *se* is in an outer sface cycle.

bool *se.in_inner_sface_cycle() const*

determines whether *se* is in an inner sface cycle.

bool *se.in_outer_facet_cycle() const*

determines whether *se* is in an outer facet cycle.

bool *se.in_inner_facet_cycle() const*

determines whether *se* is in an inner facet cycle.

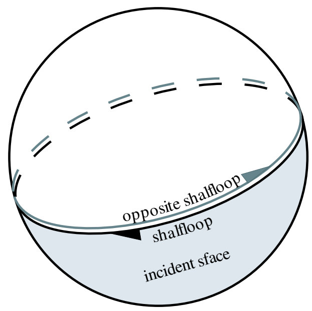
See Also

CGAL::Nef_polyhedron_3<Traits>::Halfedge . . . . page 2021
CGAL::Nef_polyhedron_3<Traits>::Halffacet . . . . page 2023
CGAL::Nef_polyhedron_3<Traits>::SFace . . . . page 2031
CGAL::Nef_polyhedron_S2<Traits>::Sphere_circle . . . . page 1611

CGAL::Nef_polyhedron_3<Traits>::SHalfloop

Definition

A shalfloop is a great circle on a sphere map. Figure 29.9 depicts the relationship between a shalfloop and its incident shalfloops, and sfaces on a sphere map. A shalfloop is an oriented sloop. It is always paired with a shalfloop whose supporting *Sphere_circle* is pointing in the opposite direction. The *twin()* member function returns this shalfloop of opposite orientation.



A sphere map having a shalfloop models the neighborhood of a vertex which is isolated on a facet. That facet is returned by the member function *facet*.

#include <CGAL/Nef_polyhedron_3.h>

Types

The following types are the same as in *Nef_polyhedron_3<Traits>*.

SHalfloop:: Mark	type of mark.
SHalfloop:: Sphere_circle	sphere circle type stored in SHalfloop.
SHalfloop:: Halffacet_const_handle	const handle to Halffacet.
SHalfloop:: SHalfloop_const_handle	const handle to SHalfloop.
SHalfloop:: SFace_const_handle	const handle to SFace.

Creation

There is no need for a user to create a *SHalfloop* explicitly. The class *Nef_polyhedron_3<Traits>* manages the needed shalfloops internally.

Operations

Mark	*se.mark() const*	the mark of *se* .
Sphere_circle	*se.circle() const*	the sphere circle of *se* .

SHalfloop_const_handle	*se.twin() const*	the twin of *se* .
Halffacet_const_handle	*se.facet() const*	the facet that corresponds to *se* in the 3D incidence structure.
SFace_const_handle	*se.incident_sface() const*	the incident sface of *se* .

See Also

CGAL::Nef_polyhedron_3<Traits>::Halffacet . . . . page 2023
CGAL::Nef_polyhedron_3<Traits>::SFace . . . . page 2031
CGAL::Nef_polyhedron_S2<Traits>::Sphere_point . . . . page 1608

CGAL::Nef_polyhedron_3<Traits>::SFace

Class

Definition

Figures 29.9 and 29.9 illustrate the incidences of an sface. An sface is described by its boundaries. An entry item to each boundary cycle can be accessed using the iterator range (*sface_cycles_begin()*/*sface_cycles_end()*). Additionally, *Nef_polyhedron_S2* provides the macro *CGAL_forall_sface_cylces_of*. The iterators are of type *SFace_cycle_const_iterator* and represent either a shalfedge, a shalfloop, or a svertex.

#include <CGAL/Nef_polyhedron_3.h>

Types

The following types are the same as in *Nef_polyhedron_3<Traits>*.

SFace:: Mark	type of mark.
SFace:: Object_list	list of Object handles.
SFace:: Vertex_const_handle	const handle to Vertex.
SFace:: Volume_const_handle	const handle to Volume.
SFace:: SFace_const_handle	const handle to SFace.
SFace:: SFace_cycle_const_iterator	const iterator over the entries to all sface cycles of a sface.

Creation

There is no need for a user to create a *SFace* explicitly. The class *Nef_polyhedron_3<Traits>* manages the needed sfaces internally.

Operations

Mark	*sf.mark() const*	the mark of *sf* .
Vertex_const_handle	*sf.center_vertex() const*	the center vertex of the sphere map *sf* belongs to.
Volume_const_handle	*sf.volume() const*	the volume that corresponds to *sf* in the 3D incidence structure.
SFace_cycle_const_iterator	*sf.sface_cycle_begin() const*	iterator over the entries to all sface cycles of *sf* .

SFace_cycle_const_iterator *sf.sface_cycle_end() const*

past-the-end iterator.

See Also

CGAL::Nef_polyhedron_3<Traits>::Vertex . . . . page 2020
CGAL::Nef_polyhedron_3<Traits>::Volume . . . . page 2025

CGAL::Nef_polyhedron_3<Traits>::Halffacet_cycle_iterator

Class

Definition

The type *Halffacet_cycle_iterator* iterates over a list of *Object_handles*. Each item of that list can either be assigned to *SHalfedge_handle* or *SHalfloop_handle*. To find out which of these assignment works out, the member functions *is_shalfedge()* and *is_shalfloop()* are provided.

#include <CGAL/Nef_polyhedron_3.h>

Types

Halffacet_cycle_iterator:: SHalfedge_handle	const handle to SHalfedge.
Halffacet_cycle_iterator:: SHalfloop_handle	const handle to SHalfloop.

Creation

Halffacet_cycle_iterator hfc;	default constructor.

Operations

bool	*hfc.is_shalfedge() const*	returns true if *hfc* represents a *SHalfedge_handle*.
bool	*hfc.is_shalfloop() const*	returns true if *hfc* represents a *SHalfloop_handle*.
SHalfedge_handle	*SHalfedge_handle(hfc)*	casts *hfc* to *SHalfedge_handle*.
SHalfloop_handle	*SHalfloop_handle(hfc)*	casts *hfc* to *SHalfloop_handle*.

See Also

CGAL::Nef_polyhedron_3<Traits>::SHalfedge . . . . page 2026
CGAL::Nef_polyhedron_3<Traits>::SHalfloop . . . . page 2029

CGAL::Nef_polyhedron_3<Traits>::SFace_cycle_iterator

Definition

The type *SFace_cycle_iterator* iterates over a list of *Object_handles*. Each item of that list can either be assigned to *SVertex_handle*, *SHalfedge_handle* or *SHalfloop_handle*. To find out which of these assignment works out, the member functions *is_svertex()*, *is_shalfedge()* and *is_shalfloop()* are provided.

#include <CGAL/Nef_polyhedron_3.h>

Types

SFace_cycle_iterator:: SVertex_handle	const handle to SVertex.
SFace_cycle_iterator:: SHalfedge_handle	const handle to SHalfedge.
SFace_cycle_iterator:: SHalfloop_handle	const handle to SHalfloop.

Creation

SFace_cycle_iterator sfc;	default constructor.

Operations

bool	*sfc.is_svertex() const*	returns true if *sfc* represents a *SVertex_handle*.
bool	*sfc.is_shalfedge() const*	returns true if *sfc* represents a *SHalfedge_handle*.
bool	*sfc.is_shalfloop() const*	returns true if *sfc* represents a *SHalfloop_handle*.
SVertex_handle	*SVertex_handle(sfc)*	casts *sfc* to *SVertex_handle*.
SHalfedge_handle	*SHalfedge_handle(sfc)*	casts *sfc* to *SHalfedge_handle*.
SHalfloop_handle	*SHalfloop_handle(sfc)*	casts *sfc* to *SHalfloop_handle*.

See Also

CGAL::Nef_polyhedron_3<Traits>::Halfedge . . . . page 2021
CGAL::Nef_polyhedron_3<Traits>::SHalfedge . . . . page 2026
CGAL::Nef_polyhedron_3<Traits>::SHalfloop . . . . page 2029

CGAL::OFF_to_nef_3

Definition

This constructor creates a 3D Nef polyhedron from OFF file. OFF file is read from input stream *in*. The purpose of *OFF_to_nef_3* is to create a Nef polyhedron from an OFF file that cannot be handled by the *Nef_polyhedron_3* constructors. It handles double coordinates while using a homogenous kernel, non-coplanar facets, surfaces with boundaries, self-intersecting surfaces, and single facets. Every closed volume gets marked. The function returns the number of facets it could not handle.

template<class Nef_polyhedron_3>
std::size_t *OFF_to_nef_3(std::istream& in, Nef_polyhedron_3& N)*

See Also

CGAL::Nef_polyhedron_3<Traits> . . . . page 2013

Class

CGAL::Nef_nary_union_3<Nef_polyhedron_3>

Definition

This class helps to perform the union of a set of 3D Nef polyhedra efficiently. It succesively applies the binary union operation of *Nef_polyhedron_3*, but schedules these union operations in an opportune way. The class is most efficient, if the polyhedra are added in sorted order. Any order that reflects proximity in the three-dimensional space is helpful. To allow saving memory space, the sorting is left to the user. This way the user can generate the polyhedra in a sorted way and add them one by one to *Nef_nary_union_3*.

#include <CGAL/Nef_nary_union_3.h>

Parameters

template < class Nef_polyhedron_3,
class Nef_nary_union_3;

As a template parameter an instantiation of the template class Nef polyhedra is needed.

Creation

Nef_nary_union_3<Nef_polyhedron_3> U;	initialization only.

Member Functions

Nef_polyhedron_3 *U.get_union() const*

returns the union of the polyhedra previously added to the class.

void *U.add_polyhedron(Nef_polyhedron_3 N)*

adds a polyhedron.

CGAL::Nef_polyhedron_3<Traits> . . . . page 2013

CGAL::operator<<

Definition

This operator writes The Nef polyhedron *N* to the output stream *out* using a propriatary file format. It includes the complete incidence structure, the geometric data, and the marks of each item.

Using CGAL stream modifiers the following output formats can be chosen: ASCII(*set_ascii_mode*), binary(*set_binary_mode*) or pretty(*set_pretty_mode*). The mandatory format is the ASCII format. It is recommended to use this format for file input and output.

As the output depends on the output operators of the geometric primitives provided by the traits class, it might not be possible that the input operator and output operators of different traits classes are not compatible. We recommend to use the CGAL kernels *Homogeneous*, *Simple_homogeneous*, or *Extended_homogeneous* parametrized with any exact number type that models $\mathbb{Z}$ (e.g. *Gmpz* or *leda_integer*).

A bounded *Nef_polyhedron_3<Extended_homogeneous>* is automatically written as though *Nef_polyhedron_3<CGAL::Homogeneous>* or *Nef_polyhedron_3<CGAL::Simple_homogeneous>* is used. As a result, the input operator of each of these types can read the output.

#include <CGAL/IO/Nef_polyhedron_iostream_3.h>

template <class Traits>
ostream& *ostream& out << CGAL::Nef_polyhedron_3<Traits> N*

See Also

CGAL::Nef_polyhedron_3<Traits> . . . . page 2013
operator>> . . . . page 2038

CGAL::operator>>

Definition

This operator reads a Nef polyhedron, which is given in the proprietary file format written by the input operator *in* and assigns it to *N*. It includes the complete incidence structure, the geometric data, and the marks of each item.

It is recommended to use the CGAL kernels *Homogeneous*, *Simple_homogeneous*, or *Extended_homogeneous* parametrized with any exact number type that models $\mathbb{Z}$ (e.g.*Gmpz* or *leda_integer*). The input and output iterators of Nef polyhedra parametrized with either of these kernels are compatible as long as the Nef polyhedron is bounded. An unbounded Nef polyhedron can only be read by a Nef polyhedron parametrized with an extended kernel. It is also recommended to use the CGAL stream modifier *set_ascii_mode*.

#include <CGAL/IO/Nef_polyhedron_iostream_3.h>

template <class Traits>
istream& *istream& in >> CGAL::Nef_polyhedron_3<Traits>& N*

See Also

CGAL::Nef_polyhedron_3<Traits> . . . . page 2013
operator<< . . . . page 2037

CGAL::Qt_widget_Nef_3<Nef_polyhedron_3>

Definition

The class *Qt_widget_Nef_3* uses the OpenGL interface of Qt to display a *Nef_polyhedron_3*. Its purpose is to provide an easy to use viewer for *Nef_polyhedron_3*. There are no means provided to enhance the functionality of the viewer.

In addition to the functions inherited from the Qt class *OGLWidget* via, *Qt_widget_Nef_3* only has a single public constructor. For the usage of *Qt_widget_Nef_3* see the example below.

#include <CGAL/IO/Qt_widget_Nef_3.h>

Parameters

The template parameter expects an instantiation of *Nef_polyhedron_3<Traits>*.

Creation

Qt_widget_Nef_3<Nef_polyhedron_3> W(Nef_polyhedron_3 N);

Creates a widget *W* for displaying the 3D Nef polyhedron N.

See Also

CGAL::Nef_polyhedron_3<Traits> . . . . page 2013

Example

This example reads a 3D Nef polyhedron from standard input and displays it in a Qt widget.

```
#include <CGAL/basic.h>
#include <CGAL/Gmpz.h>
#include <CGAL/Homogeneous.h>
#include <CGAL/Nef_polyhedron_3.h>
#include <CGAL/IO/Nef_polyhedron_iostream_3.h>
#include <CGAL/IO/Qt_widget_Nef_3.h>
#include <qapplication.h>

typedef CGAL::Homogeneous<CGAL::Gmpz> Kernel;
typedef CGAL::Nef_polyhedron_3<Kernel> Nef_polyhedron_3;

int main(int argc, char* argv[]) {
  Nef_polyhedron_3 N;
  std::cin >> N;

  QApplication a(argc, argv);
```

```
  CGAL::Qt_widget_Nef_3<Nef_polyhedron_3>* w =
    new CGAL::Qt_widget_Nef_3<Nef_polyhedron_3>(N);
  a.setMainWidget(w);
  w->show();
  return a.exec();
}
```

File: demo/Nef_3/visualization_SNC.cpp

Chapter 30

Convex Decomposition of Polyhedra

Peter Hachenberger

30.1 Introduction

For many applications on non-convex polyhedra, there are efficient solutions that first decompose the polyhedron into convex pieces. As an example, the Minkowski sum of two polyhedra can be computed by decomposing both polyhedra into convex pieces, compute pair-wise Minkowski sums of the convex pieces, and unite the pair-wise sums.

While it is desirable to have a decomposition into a minimum number of pieces, this problem is known to be NP-hard [Cha84]. Our implementation decomposes a Nef polyhedron N into $O(r^2)$ convex pieces, where r is the number of edges that have two adjacent facets that span an angle of more than 180 degrees with respect to the interior of the polyhedron. Those edges are also called reflex edges. The bound of $O(r^2)$ convex pieces is worst-case optimal [Cha84].

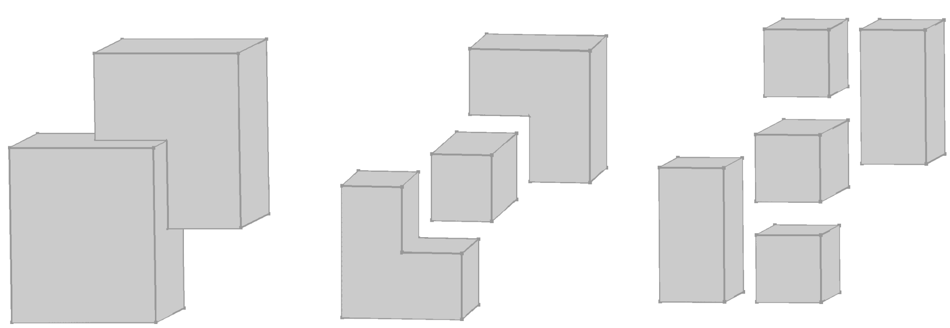


Figure 30.1: Vertical decomposition based on the insertion of vertical facets (viewed from the top). Left: Non-convex polyhedron. Middle: Non-vertical reflex edges have been resolved. Right: Vertical reflex edges have been resolved. The sub-volumes are convex.

Our decomposition runs in two steps. In the first step, each non-vertical reflex edge e is resolved by insertion of vertical facets through e. In the second step, we do the same with the vertical reflex edges. Figure 30.1 illustrates the two steps.

At the moment our implementation is restricted to the decomposition of bounded polyhedra. An extension to unbounded polyhedra is planned.

30.2 Interface and Usage

An instance of *Nef_polyhedron_3* represents a subdivision of the three-dimensional space into vertices, edges, facets, and volumes. Some of these items form the polyhedron (selected), while others represent the outer volume or holes within the polyhedron (unselected). As an example, the unit cube is the point set $[0,1]^3$. The smallest subdivision that represents the unit cube has 8 vertices, 12 edges, 6 facets, and 2 volumes. The volumes enclosed by the vertices, edges, and facets is the interior of the cube and therefore selected. The volume outside the cube does not belong to it and is therefore unselected. The vertices, edges, and facets—also denoted as boundary items—are needed to separate the two volumes, but are also useful for representing topological properties. In case of the (closed) unit cube the boundary items are part of the polyhedron and therefore selected, but in case of the open unit cube $[0,1)^3$ they are unselected. Each item has its own selection mark, which allows the correct representation of Nef polyhedra, which are closed under Boolean and topological operations. Details can be found in the chapter on 3D Boolean operations on Nef polyhedra 29.

Usually, an instance of *Nef_polyhedron_3* does not contain any redundant items. However, the function *convex_decomposition_3* subdivides selected volumes of a given *Nef_polyhedron_3* by selected facets. These additional facets are therefore redundant, i.e., their insertion alters the representation of the polyhedron, but not the polyhedron itself.

When *convex_decomposition_3* resolved all reflex edges, the selected sub-volumes have become convex. Each of them is represented by a separate volume item and can therefore be traversed separately 29.7.2. Another possibility of accessing the convex pieces is to convert them into separate Nef polyhedra, as illustrated by the example code given below.

Note that due to the restriction to bounded polyhedra, the use of extended kernels is unnecessary and expensive. We therefore do not support the use of extended kernels in the convex decomposition (see Chapter 29).

```
#include <CGAL/Exact_predicates_exact_constructions_kernel.h>
#include <CGAL/Polyhedron_3.h>
#include <CGAL/Nef_polyhedron_3.h>
#include <CGAL/IO/Nef_polyhedron_iostream_3.h>
#include <CGAL/Nef_3/SNC_indexed_items.h>
#include <CGAL/convex_decomposition_3.h>
#include <list>

typedef CGAL::Exact_predicates_exact_constructions_kernel Kernel;
typedef CGAL::Polyhedron_3<Kernel> Polyhedron_3;
typedef CGAL::Nef_polyhedron_3<Kernel, CGAL::SNC_indexed_items>
   Nef_polyhedron_3;
typedef Nef_polyhedron_3::Volume_const_iterator Volume_const_iterator;

int main() {

  Nef_polyhedron_3 N;
  std::cin >> N;

  CGAL::convex_decomposition_3(N);
  std::list<Polyhedron_3> convex_parts;

  // the first volume is the outer volume, which is
```

```
  // ignored in the decomposition
  Volume_const_iterator ci = ++N.volumes_begin();
  for( ; ci != N.volumes_end(); ++ci) {
    if(ci->mark()) {
      Polyhedron_3 P;
      N.convert_inner_shell_to_polyhedron(ci->shells_begin(), P);
      convex_parts.push_back(P);
    }
  }
  std::cout << "decomposition into " << convex_parts.size() << " convex
   parts " << std::endl;

}
```

File: examples/Convex_decomposition_3/list_of_convex_parts.cpp

Convex Decomposition of Polyhedra Reference Manual

Peter Hachenberger

The function *convex_decomposition_3* takes a *Nef_polyhedron_3 N* as input parameter and inserts additional facets, such that each bounded marked volume (the outer volume is unbounded) is subdivided into convex pieces.

30.3 Classified Reference Pages

Functions

void *convex_decomposition_3(Nef_polyhedron_3& N)*

page 2046

30.4 Alphabetical List of Reference Pages

Function

CGAL::convex_decomposition_3

Definition

The function *convex_decomposition_3* inserts additional facets into the given *Nef_polyhedron_3* N, such that each bounded marked volume (the outer volume is unbouned) is subdivided into convex pieces. The modified polyhedron represents a decomposition into $O(r^2)$ convex pieces, where r is the number of edges that have two adjacent facets that span an angle of more than 180 degrees with respect to the interior of the polyhedron.

The worst-case running time of our implementation is $O(n^2 r^4 \sqrt[3]{nr^2} \log(nr))$, where n is the complexity of the polyhedron (the complexity of a *Nef_polyhedron_3* is the sum of its *Vertices*, *Halfedges* and *SHalfedges*) and r is the number of reflex edges.

void *convex_decomposition_3(Nef_polyhedron_3& N)*

Precondition:

The polyhedron N is bounded. Otherwise, the outer volume is ignored.

Postcondition:

If the polyhedron N is non-convex, it is modified to represent the convex decomposition. If N is convex, it is not modified.

See Also

CGAL::Nef_polyhedron_3<Traits> .... page 2013

Chapter 31

3D Minkowski Sum of Polyhedra

Peter Hachenberger

31.1 Introduction

Figure 31.1: The Minkowski sum of a spoon and a star.

The Minkowski sum of two point sets P and Q in $\mathbb{R}^d$, denoted by $P \oplus Q$, is defined as the set $\{p+q : p \in P, q \in Q\}$. Minkowski sums are used in a wide range of applications such as robot motion planning [Lat91] and computer-aided design [EK99]. Figure 31.2 shows an example how Minkowski sums can be used to plan the motion of a translational robot. We want to know which are legal positions of the robot, and where can the robot go to from a specified starting position. If we model both the robot and the obstacles as a polyhedron and compute the Minkowski sum of the inverted robot (robot reflected at the origin) and the obstacles, then this Minkowski sum represents all illegal positions of the robot, i.e., all positions where it intersects the obstacle. Of course, the complement of that polyhedron describes all legal positions of the robot, i.e., all positions where it does not intersect an obstacle.

The Minkowski sum can be illustrated as follows. Pick an arbitrary reference point r of P (black dot in the lower left corner of the robot in Figure 31.2). Then place the inverted set $-P$ on Q, such that $-r$ is on the boundary of Q. Finally, move $-P$ along the complete boundary of Q. The union of Q and the points swept by $-P$ is the Minkowski sum of P and Q.

Implementing the Minkowski sum, the reference point does not need to be chosen. It is implicitly given as the origin of the coordinate system. Choosing a different reference point is equivalent to translating the coordinate system. Such a translation does not change the shape of the Minkowski sum; it only translates the Minkowski sum by the same vector.

This package provides a function *minkowski_sum_3* that computes the Minkowski sum of two Nef polyhedra.

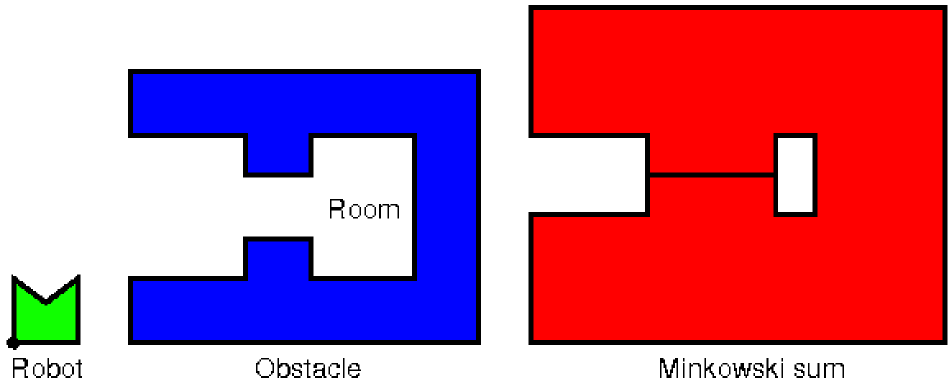


Figure 31.2: Can the robot enter the room? The Minkowski sum of the inverted robot and the obstacle describes the illegal positions of the robot with respect to the obstacle. Since the boundary of the Minkowski sum describes legal positions, there is a path for the robot between the outer area and the room.

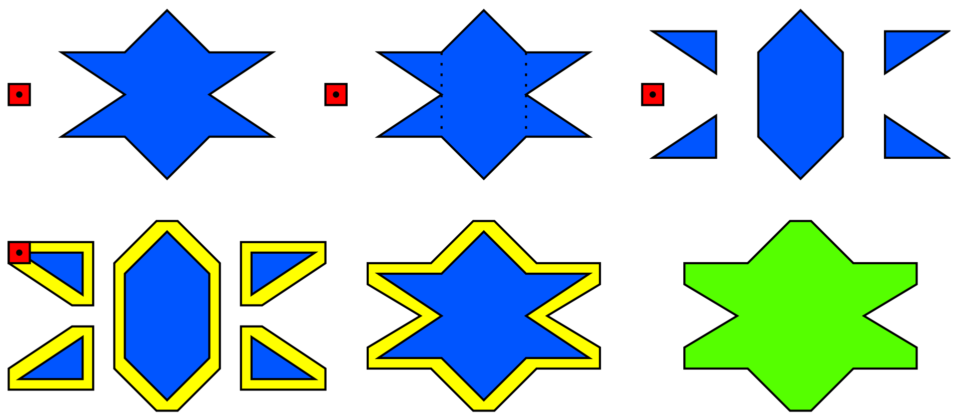


Figure 31.3: The decomposition method decomposes both input polyhedra into convex parts, computes all pairwise Minkowski sums of the convex parts, and merges the pairwise sums.

We do not support arbitrary Nef polyhedra, yet. The restrictions are discussed in detail in Section 31.3.

31.2 Decomposition Method

The decomposition method for computing the Minkowski sum of non-convex polyhedra makes use of the fact that Minkowski sums of convex polyhedra are rather easy to compute. It decomposes both polyhedra into convex pieces, computes all pairwise Minkowski sums of the convex pieces, and merges the pairwise sums [dBvKOS97].

Minkowski sum are inherently complex to compute. Using the decomposition method, each polyhedron might be divided into a quadratic number of pieces, which is worst-case optimal. Then up to n^2m^2 pairwise sums have to be computed and merged, where n and m are the complexities of the two input polyhedra (the complexity of a *Nef_polyhedron_3* is the sum of its *Vertices*, *Halfedges* and *SHalfedges*). In total the operation runs in $O(n^3m^3)$ time.

Since the computation of the Minkowski sum takes quite some time, we give the running times of some

Minkowski sum computations. They were computed with CGAL 3.3 on a machine with a 2.4 GHz AMD Opteron processor and 4 GB RAM. The code was compiled with g++ 3.2 and compiler options -O2. The *Nef_polyhedron_3* class was instantiated with the geometric kernel *Homogeneous<leda_integer>*. The Minkowski sum of the spoon and the star is illustrated in Figure 31.1.

model 1			model 2			running
name	facets	conv. pcs.	name	facets	conv. pcs.	time
mushroom	448	255	cube	6	1	204s
mushroom	448	255	ball1	128	1	553s
spoon	336	186	star	24	5	882s
cup	1000	774	ball2	1000	1	9851s

Table 31.1: Performance of the function *minkowski_sum_3*.

31.3 Features and Restrictions

This package was written to allow the computation of Minkowski sums of full-dimensional polyhedra even in so-called tight-passage scenarios. Tight passage scenarios occur in robot motion planning, when a robot is just as wide as a passage it needs to traverse. In these scenarios at least one polyhedron—the obstacles or the robot—must be modeled as an open set. Then the Minkowski sum will also be an open set and tight passages will occur as lower-dimensional exclusions, i.e., as facets, lines, or vertices that are, in contrast to the volume around them, not part of the resulting point set. Figure 31.2 shows such a tight passage scenario.

Our implementation uses *Nef_polyhedron_3* to model the input polyhedra and the result polyhedron. An instance of *Nef_polyhedron_3* represents a subdivision of the three-dimensional space into vertices, edges, facets, and volumes. Some of these items form the polyhedron (selected), while others represent the outer volume or holes within the polyhedron (unselected). As an example, the unit cube is the point set $[0,1]^3$. The smallest subdivision that represents the unit cube has 8 vertices, 12 edges, 6 facets, and 2 volumes. The volumes enclosed by the vertices, edges, and facets is the interior of the cube and therefore selected. The volume outside the cube does not belong to it and is therefore unselected. The vertices, edges, and facets—also denoted as boundary items—are needed to separate the two volumes, but are also useful for representing topological properties. In case of the (closed) unit cube the boundary items are part of the polyhedron and therefore selected, but in case of the open unit cube $[0,1)^3$ they are unselected. Each item has its own selection mark, which allows the correct representation of Nef polyhedra, which are closed under Boolean and topological operations. Details can be found in the chapter on 3D Boolean operations on Nef polyhedra 29.

The use of *Nef_polyhedron_3* allows many scenarios beyond the Minkowski sum of two solids. First, they can model the input and the result of a tight passage scenario, i.e., they can model open and closed solids as is needed for the input models, and they can model tight passages, which are lower-dimensional exclusions represented as unselected facets, edges, and vertices. We strive for extending the package to work for arbitrary 3D Nef polyhedra. In addition to the Minkowski sums of two solids, we added several features. At the moment we allow an input polyhedron to consist of:

1. singular vertices
2. singular edges
3. singular convex facets without holes
4. surfaces with convex facets that have no holes.
5. three-dimensional features, whose coplanar facets have common selection marks (this includes open and closed solids)

Taking a different viewpoint, the implementation is restricted as follows:

1. The input polyhedra must be bounded (selected outer volume is ignored).

2. All sets of coplanar facets of a full-dimensional feature must have the same selection mark (in case of different selection marks, unselected is assumed).

3. All facets of lower-dimensional features need to be convex and must not have holes (non-convex facets and holes are ignored).

The second restriction might seem a bit odd. It stems from the fact that the Minkowski sum on convex polyhedra can only handle polyhedra, whose sides consist of a single facet. The decomposition process usually yields complex adjacency relations between a convex part, its adjacent convex parts, and the outer volume. The side of a convex piece is then decomposed into several facets, each of which represents one of these adjacency relations. For the convex Minkowski sum, we ignore the decompositions of the sides, but need to find a common selection mark. If there are two facets that are adjacent to the outer volume, but have different selections marks, we cannot set a common selection mark without spoiling the correctness of the Minkowski sum.

31.4 Usage

The function *minkowski_sum_3* should be used with the *Exact_predicates_exact_constructions_kernel*, which often is the most efficient choice and allows floating-point input. Consult Section for more details.

The following example code illustrates the usage of the function *minkowski_sum_3*. Note that either input polyhedra will be modified by the function if it is non-convex. So, if they are needed further on, they need to be copied, first. The copying is not done by the function itself to keep the memory usage as small as possible.

```
#include <CGAL/Exact_predicates_exact_constructions_kernel.h>
#include <CGAL/Nef_polyhedron_3.h>
#include <CGAL/IO/Nef_polyhedron_iostream_3.h>
#include <CGAL/minkowski_sum_3.h>
#include <fstream>
#include <iostream>

typedef CGAL::Exact_predicates_exact_constructions_kernel Kernel;
typedef CGAL::Nef_polyhedron_3<Kernel> Nef_polyhedron;

int main() {

  Nef_polyhedron N0, N1;
  std::ifstream in("cube.nef3");
  in >> N0;
  std::cin >> N1;
  Nef_polyhedron result = CGAL::minkowski_sum_3(N0, N1);
}
```

File: examples/Minkowski_sum_3/cube_offset.cpp

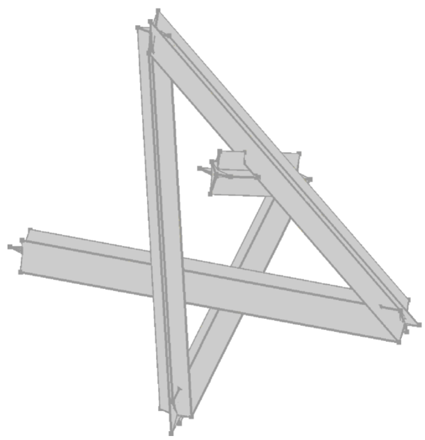


Figure 31.4: The region swept by a star that moves along a polygonal path.

31.5 Glide

With the function *minkowski_sum_3* it is also possible to realize other interesting geometric operations like the glide operation, which computes the point set swept by a polyhedron that moves along a polygonal path. The following example shows how to construct a polygonal path and then compute the glide operation by calling the function *minkowski_sum_3*.

```
#include <CGAL/Exact_predicates_exact_constructions_kernel.h>
#include <CGAL/Nef_polyhedron_3.h>
#include <CGAL/IO/Nef_polyhedron_iostream_3.h>
#include <CGAL/minkowski_sum_3.h>
#include <iostream>

typedef CGAL::Exact_predicates_exact_constructions_kernel Kernel;
typedef CGAL::Nef_polyhedron_3<Kernel>     Nef_polyhedron;
typedef Kernel::Point_3 Point_3;
typedef Point_3* point_iterator;
typedef std::pair<point_iterator,point_iterator>
  point_range;
typedef std::list<point_range> polyline;

int main(int argc, char* argv[])
{
  Nef_polyhedron N0;
  std::cin >> N0;
  Point_3 pl[6] =
```

```
    {Point_3(-100,0,0),
     Point_3(40,-70,0),
     Point_3(40,50,40),
     Point_3(-90,-60,60),
     Point_3(0, 0, -100),
     Point_3(30,0,150)
  };

  polyline poly;
  poly.push_back(point_range(pl,pl+6));
  Nef_polyhedron N1(poly.begin(), poly.end(),
   Nef_polyhedron::Polylines_tag());
  Nef_polyhedron result = CGAL::minkowski_sum_3(N0, N1);
}
```

File: examples/Minkowski_sum_3/glide.cpp

3D Minkowski sum of Polyhedra Reference Manual

Peter Hachenberger

The function *minkowski_sum_3* computes the Minkowski sum of two given 3D Nef polyhedra.

31.6 Classified Reference Pages

Functions

31.7 Alphabetical List of Reference Pages

Function

CGAL::minkowski_sum_3

Definition

The function *minkowski_sum_3* computes the Minkowski sum of two given 3D Nef polyhedra $N0$ and $N1$. Note that the function runs in $O(n^3 m^3)$ time in the worst case, where n and m are the complexities of the two input polyhedra (the complexity of a *Nef_polyhedron_3* is the sum of its *Vertices*, *Halfedges* and *SHalfedges*).

Nef_polyhedron_3   *minkowski_sum_3(Nef_polyhedron_3 N0, Nef_polyhedron_3 N1)*

Precondition:

An input polyhedron may consist of:

1. singular vertices
2. singular edges
3. singular convex facets without holes
4. surfaces with convex facets that have no holes.
5. three-dimensional features, whose coplanar facets have common selection marks (this includes open and closed solids)

Taking a different viewpoint, the implementation is restricted as follows:

1. The input polyhedra must be bounded (selected outer volume is ignored).
2. All sets of coplanar facets of a full-dimensional feature must have the same selection mark (in case of different selection marks, unselected is assumed).
3. All facets of lower-dimensional features need to be convex and must not have holes (non-convex facets and holes are ignored).

Postcondition:

If either of the input polyhedra is non-convex, it is modified during the computation, i.e., it is decomposed into convex pieces.

See Also

CGAL::Nef_polyhedron_3<Traits> . . . . page 2013
CGAL::convex_decomposition_3 . . . . page 2046

Part VIII

Arrangements

Chapter 32

2D Arrangements

Ron Wein, Eric Berberich, Efi Fogel, Dan Halperin, Michael Hemmer, Oren Salzman, and Baruch Zukerman

Contents

32.1 Introduction

Given a set $\mathcal{C}$ of planar curves, the *arrangement* $\mathcal{A}(\mathcal{C})$ is the subdivision of the plane into zero-dimensional, one-dimensional and two-dimensional cells, called *vertices*, *edges* and *faces*, respectively induced by the curves in $\mathcal{C}$. Arrangements are ubiquitous in the computational-geometry literature and have many applications; see, e.g., [AS00, Hal04].

The curves in $\mathcal{C}$ can intersect each other (a single curve may also be self-intersecting or may be comprised of several disconnected branches) and are not necessarily x-monotone.[1] We construct a collection $\mathcal{C}''$ of x-monotone subcurves that are pairwise disjoint in their interiors in two steps as follows. First, we decompose each curve in $\mathcal{C}$ into maximal x-monotone subcurves (and possibly isolated points), obtaining the collection $\mathcal{C}'$. Note that an x-monotone curve cannot be self-intersecting. Then, we decompose each curve in $\mathcal{C}'$ into maximal connected subcurves not intersecting any other curve (or point) in $\mathcal{C}'$. The collection $\mathcal{C}''$ may also contain isolated points, if the curves of $\mathcal{C}$ contain such points. The arrangement induced by the collection $\mathcal{C}''$ can be conveniently embedded as a planar graph, whose vertices are associated with curve endpoints or with isolated points, and whose edges are associated with subcurves. It is easy to see that $\mathcal{A}(\mathcal{C}) = \mathcal{A}(\mathcal{C}'')$. This graph can be represented using a *doubly-connected edge list* data-structure (DCEL for short), which consists of containers of vertices, edges and faces and maintains the incidence relations among these objects.

The main idea behind the DCEL data-structure is to represent each edge using a pair of directed *halfedges*, one going from the xy-lexicographically smaller (left) endpoint of the curve toward its the xy-lexicographically larger (right) endpoint, and the other, known as its *twin* halfedge, going in the opposite direction. As each halfedge is directed, we say it has a *source* vertex and a *target* vertex. Halfedges are used to separate faces, and to connect vertices (with the exception of *isolated vertices*, which are unconnected).

If a vertex v is the target of a halfedge e, we say that v and e are *incident* to each other. The halfedges incident to a vertex v form a circular list oriented in a clockwise order around this vertex. (An isolated vertex has no incident halfedges.)

[1] A continuous planar curve C is *x-monotone* if every vertical line intersects it at most once. For example, a non-vertical line segment is always x-monotone and so is the graph of any continuous function $y = f(x)$. For convenience, we treat vertical line segments as *weakly x-monotone*, as there exists a single vertical line that overlaps them. A circle of radius r centered at (x_0, y_0) is not x-monotone, as the vertical line $x = x_0$ intersects it at $(x_0, y_0 - r)$ and at $(x_0, y_0 + r)$.

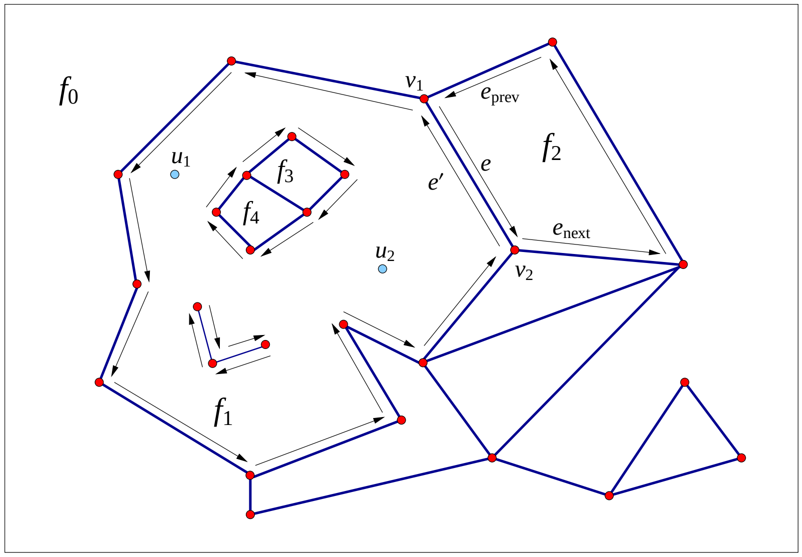


Figure 32.1: An arrangement of interior-disjoint line segments with some of the DCEL records that represent it. The unbounded face f_0 has a single connected component that forms a hole inside it, and this hole is comprised if several faces. The half-edge e is directed from its source vertex v_1 to its target vertex v_2. This edge, together with its twin e', correspond to a line segment that connects the points associated with v_1 and v_2 and separates the face f_1 from f_2. The predecessor e_{prev} and successor e_{next} of e are part of the chain that form the outer boundary of the face f_2. The face f_1 has a more complicated structure as it contains two holes in its interior: One hole consists of two adjacent faces f_3 and f_4, while the other hole is comprised of two edges. f_1 also contains two isolated vertices u_1 and u_2 in its interior.

Each halfedge e stores a pointer to its *incident face*, which is the face lying to its left. Moreover, every halfedge is followed by another halfedge sharing the same incident face, such that the target vertex of the halfedge is the same as the source vertex of the next halfedge. The halfedges are therefore connected in circular lists, and form chains, such that all edges of a chain are incident to the same face and wind along its boundary. We call such a chain a *connected component of the boundary* (or *CCB* for short).

The unique CCB of halfedges winding in a counterclockwise orientation along a face boundary is referred to as the *outer CCB* of the face. For the time being let us consider only arrangements of bounded curves, such that exactly one unbounded face exists in every arrangement. The unbounded face does not have an outer boundary. Any other connected component of the boundary of the face is called a *hole* (or *inner CCB*), and can be represented as a circular chain of halfedges winding in a clockwise orientation around it. Note that a hole does not necessarily correspond to a single face, as it may have no area, or alternatively it may consist of several connected faces. Every face can have several holes contained in its interior (or no holes at all). In addition, every face may contain isolated vertices in its interior. See Figure 32.1 for an illustration of the various DCEL features. For more details on the DCEL data structure see [dBvKOS00, Chapter 2].

The x-monotone curves of an arrangement are embedded in an rectangular two-dimensional area called the parameter space.[2] The parameter space is defined as $X \times Y$, where X and Y are open, half-open, or closed intervals with endpoints in the compactified real line $\mathbb{R} \cup \{-\infty, +\infty\}$. Let b_l, b_r, b_b, and b_t denote the endpoints of X and Y, respectively. We typically refer to these values as the left, right, bottom, and top sides of the boundary of the parameter space. If the parameter space is, for example, the entire compactified plane, which is currently the only option supported by the package, $b_l = b_b = -\infty$ and $b_r = b_t = +\infty$.

The rest of this chapter is organized as follows: In Section 32.2 we review in detail the interface of the

[2]The term parameter space stems from a major extension the arrangement package is going through to support arrangements embedded on certain two-dimensional parametric surfaces in three-dimensions (or higher).

Arrangement_2 class-template, which is the central component in the arrangement package. In Section 32.3 we show how queries on an arrangement can be issued. In Section 32.4 we review some important free (global) functions that operate on arrangements, the most important ones being the free insertion-functions. Section 32.6 contains detailed descriptions of the various geometric traits classes included in the arrangement package. Using these traits classes it is possible to construct arrangements of different families of curves. In Section 32.7 we review the notification mechanism that allows external classes to keep track of the changes that an arrangement instance goes through. Section 32.8 explains how to extend the DCEL records, to store extra data with them, and to efficiently update this data. In Section 32.9 we introduce the fundamental operation of overlaying two arrangements. Section 32.10 describes the *Arrangement_with_history_2* class-template that extends the arrangement by storing additional history records with its curves. Finally, in Section 32.11 we review the arrangement input/output functions.

32.2 The Main Arrangement Class

The class *Arrangement_2<Traits,Dcel>* is the main class in the arrangement package. It is used to represent planar arrangements and provides the interface needed to construct them, traverse them, and maintain them. An arrangement is defined by a geometric *traits* class that determines the family of planar curves that form the arrangement, and a DCEL class, which represents the *topological structure* of the planar subdivision. It supplies a minimal set of geometric operations (predicates and constructions) required to construct and maintain the arrangement and to operate on it.

The design of the arrangement package is guided by the need to separate between the representation of the arrangements and the various geometric algorithms that operate on them, and by the need to separate between the topological and geometric aspects of the planar subdivision. The separation is exhibited by the two template parameters of the *Arrangement_2* template:

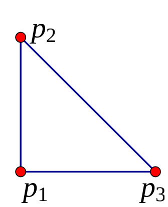
- The *Traits* template-parameter should be instantiated with a model of the *ArrangementBasicTraits_2* concept. The traits class defines the types of x-monotone curves and two-dimensional points, *X_monotone_curve_2* and *Point_2* respectively, and supports basic geometric predicates on them.

  In the first sections of this chapter we always use *Arr_segment_traits_2* as our traits class, to construct arrangements of line segments. However, the arrangement package contains several other traits classes that can handle also polylines (continuous piecewise-linear curves), conic arcs, and arcs of rational functions. We exemplify the usage of these traits classes in Section 32.6.

- The *Dcel* template-parameter should be instantiated with a class that is a model of the *ArrangementDcel* concept. The value of this parameter is *Arr_default_dcel<Traits>* by default. However, in many applications it is necessary to extend the DCEL features; see Section 32.8 for further explanations and examples.

32.2.1 A Simple Program

The simple program listed below constructs a planar map of three line segments forming a triangle. The constructed arrangement is instantiated with the *Arr_segment_traits_2* traits class to handle segments only. The resulting arrangement consists of two faces, a bounded triangular face and the unbounded face. The program is not very useful as it is, since it ends immediately after the arrangement is constructed. We give more enhanced examples in the rest of this chapter.




```
#include <CGAL/Cartesian.h>
#include <CGAL/MP_Float.h>
```

```
#include <CGAL/Quotient.h>
#include <CGAL/Arr_segment_traits_2.h>
#include <CGAL/Arrangement_2.h>

typedef CGAL::Quotient<CGAL::MP_Float>     Number_type;
typedef CGAL::Cartesian<Number_type>       Kernel;
typedef CGAL::Arr_segment_traits_2<Kernel> Traits_2;
typedef Traits_2::Point_2                  Point_2;
typedef Traits_2::X_monotone_curve_2       Segment_2;
typedef CGAL::Arrangement_2<Traits_2>      Arrangement_2;

int main()
{
  Arrangement_2   arr;
  Segment_2       cv[3];
  Point_2         p1 (0, 0), p2 (0, 4), p3 (4, 0);

  cv[0] = Segment_2 (p1, p2);
  cv[1] = Segment_2 (p2, p3);
  cv[2] = Segment_2 (p3, p1);
  CGAL::insert (arr, &cv[0], &cv[3]);

  return (0);
}
```

32.2.2 Traversing the Arrangement

The simplest and most fundamental arrangement operations are the various traversal methods, which allow users to systematically go over the relevant features of the arrangement at hand.

As mentioned above, the arrangement is represented as a DCEL, which stores three containers of vertices, halfedges and faces. Thus, the *Arrangement_2* class supplies iterators for these containers. For example, the methods *vertices_begin()* and *vertices_end()* return *Arrangement_2::Vertex_iterator* objects that define the valid range of arrangement vertices. The value type of this iterator is *Arrangement_2::Vertex*. Moreover, the vertex-iterator type is equivalent to *Arrangement_2::Vertex_handle*, which serves as a pointer to a vertex. As we show next, all functions related to arrangement features accept handle types as input parameters and return handle types as their output.

In addition to the iterators for arrangement vertices, halfedges and faces, the arrangement class also provides *edges_begin()* and *edges_end()* that return *Arrangement_2::Edge_iterator* objects for traversing the arrangement edges. Note that the value type of this iterator is *Arrangement_2::Halfedge*, representing one of the twin halfedges that represent the edge.

All iterator, circulator[3] and handle types also have non-mutable (*const*) counterparts. These non-mutable iterators are useful to traverse an arrangement without changing it. For example, the arrangement has a non-constant member function called *vertices_begin()* that returns a *Vertex_iterator* object and another const member function that returns a *Vertex_const_iterator* object. In fact, all methods listed in this section that return an iterator, a circulator or a handle have non-mutable counterparts. It should be noted that, for example, *Vertex_handle* can be readily converted into a *Vertex_const_handle*, but not vice-verse.

Conversion of a non-mutable handle to a corresponding mutable handle are nevertheless possible, and can be performed using the static function *Arrangement_2::non_const_handle()* (see, e.g., Section 32.3.1). There are

[3] A *circulator* is used to traverse a circular list, such as the list of halfedges incident to a vertex — see below.

three variant of this function, one for each type of handle.

Traversal Methods for an Arrangement Vertex

A vertex is always associated with a geometric entity, namely with a *Point_2* object, which can be obtained by the *point()* method of the *Vertex* class nested within *Arrangement_2*.

The *is_isolated()* method determines whether a vertex is isolated or not. Recall that the halfedges incident to a non-isolated vertex, namely the halfedges that share a common target vertex, form a circular list around this vertex. The *incident_halfedges()* method returns a circulator of type *Arrangement_2::Halfedge_around_vertex_circulator* that enables the traversal of this circular list in a clockwise direction. The value type of this circulator is *Halfedge*.

The following function prints all the neighbors of a given arrangement vertex (assuming that the *Point_2* type can be inserted into the standard output using the << operator). The arrangement type is the same as in the simple example above.

```
void print_neighboring_vertices (Arrangement_2::Vertex_const_handle v)
{
  if (v->is_isolated()) {
    std::cout << "The vertex (" << v->point() << ") is isolated" << std::endl;
    return;
  }

  Arrangement_2::Halfedge_around_vertex_const_circulator first, curr;
  first = curr = v->incident_halfedges();

  std::cout << "The neighbors of the vertex (" << v->point() << ") are:";
  do {
    // Note that the current halfedge is directed from u to v:
    Arrangement_2::Vertex_const_handle u = curr->source();
    std::cout << " (" << u->point() << ")";
  } while (++curr != first);
  std::cout << std::endl;
}
```

In case of an isolated vertex, it is possible to obtain the face that contains this vertex using the *face()* method.

Traversal Methods for an Arrangement Halfedge

Each arrangement edge, realized as a pair of twin halfedges, is associated with an *X_monotone_curve_2* object, which can be obtained by the *curve()* method of the *Halfedge* class nested in the *Arrangement_2* class.

The *source()* and *target()* methods return handles to the halfedge source and target vertices respectively. We can obtain a handle to the twin halfedge using the *twin()* method. From the definition of halfedges, it follows that if *he* is a halfedge handle, then:

- *he->curve()* is equivalent to *he->twin()->curve()*,
- *he->source()* is equivalent to *he->twin()->target()*, and

- *he->target()* is equivalent to *he->twin()->source()*.

Every halfedge has an incident face that lies to its left, which can be obtained by the *face()* method. Recall that a halfedge is always one link in a connected chain of halfedges that share the same incident face, known as a *CCB*. The *prev()* and *next()* methods return handles to the previous and next halfedges in the CCB respectively.

As the CCB is a circular list of halfedges, it is only natural to traverse it using a circulator. The *ccb()* method returns a *Arrangement_2::Ccb_halfedge_circulator* object for the halfedges along the CCB.

The function *print_ccb()* listed below prints all *x*-monotone curves along a given CCB (assuming that the *Point_2* and the *X_monotone_curve_2* types can be inserted into the standard output using the << operator).

```
void print_ccb (Arrangement_2::Ccb_halfedge_const_circulator circ)
{
  Ccb_halfedge_const_circulator curr = circ;
  std::cout << "(" << curr->source()->point() << ")";
  do {
    Arrangement_2::Halfedge_const_handle he = curr->handle();
    std::cout << "   [" << he->curve() << "]   "
              << "(" << he->target()->point() << ")";
  } while (++curr != circ);
  std::cout << std::endl;
}
```

Traversal Methods for an Arrangement Face

An arrangement of bounded curves always has a single unbounded face. The function *unbounded_face()* returns a handle to this face. (Note that an empty arrangement contains nothing *but* the unbounded face.)

Given a *Face* object, we can use the *is_unbounded()* method to determine whether it is unbounded. Bounded faces have an outer CCB, and the *outer_ccb()* method returns a circulator for the halfedges along this CCB. Note that the halfedges along this CCB wind in a counterclockwise orientation around the outer boundary of the face.

A face can also contain disconnected components in its interior, namely holes and isolated vertices:

- The *holes_begin()* and *holes_end()* methods return *Arrangement_2::Hole_iterator* iterators that define the range of holes inside the face. The value type of this iterator is *Ccb_halfedge_circulator*, defining the CCB that winds in a clockwise orientation around a hole.
- The *isolated_vertices_begin()* and *isolated_vertices_end()* methods return *Arrangement_2::Isolated_vertex_iterator* iterators that define the range of isolated vertices inside the face. The value type of this iterator is *Vertex*.

The function *print_face()* listed below prints the outer and inner boundaries of a given face, using the function *print_ccb()*, which was introduced in the previous subsection.

```
void print_face (Arrangement_2::Face_const_handle f)
{
  // Print the outer boundary.
  if (f->is_unbounded())
    std::cout << "Unbounded face. " << std::endl;
```

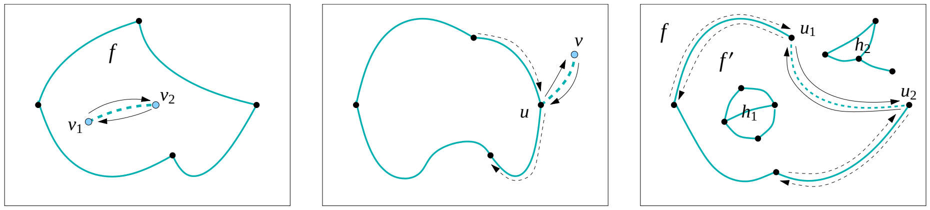
```
  else {
    std::cout << "Outer boundary: ";
    print_ccb (f->outer_ccb());
  }

  // Print the boundary of each of the holes.
  Arrangement_2::Hole_const_iterator hi;
  int                                 index = 1;
  for (hi = f->holes_begin(); hi != f->holes_end(); ++hi, ++index) {
    std::cout << "    Hole #" << index << ": ";
    print_ccb (*hi);
  }

  // Print the isolated vertices.
  Arrangement_2::Isolated_vertex_const_iterator iv;
  for (iv = f->isolated_vertices_begin(), index = 1;
       iv != f->isolated_vertices_end(); ++iv, ++index)
  {
    std::cout << "    Isolated vertex #" << index << ": "
              << "(" << iv->point() << ")" << std::endl;
  }
}
```

Additional Example

The function listed below prints the current setting of a given arrangement. This concludes the preview of the various traversal methods.[4]

```
void print_arrangement (const Arrangement_2& arr)
{
  // Print the arrangement vertices.
  Vertex_const_iterator vit;
  std::cout << arr.number_of_vertices() << " vertices:" << std::endl;
  for (vit = arr.vertices_begin(); vit != arr.vertices_end(); ++vit) {
    std::cout << "(" << vit->point() << ")";
    if (vit->is_isolated())
      std::cout << " - Isolated." << std::endl;
    else
      std::cout << " - degree " << vit->degree() << std::endl;
  }

  // Print the arrangement edges.
  Edge_const_iterator eit;
  std::cout << arr.number_of_edges() << " edges:" << std::endl;
  for (eit = arr.edges_begin(); eit != arr.edges_end(); ++eit)
    std::cout << "[" << eit->curve() << "]" << std::endl;

  // Print the arrangement faces.
  Face_const_iterator fit;
```

[4]The file *arr_print.h*, which can be found under the examples folder, includes this function and the rest of the functions listed in this section. Over there they are written in a more generic fashion, where the arrangement type serves as a template parameter for these functions, so different instantiations of the *Arrangement_2<Traits,Dcel>* template can be provided to the same function templates.



(a) (b) (c)

Figure 32.2: The various specialized insertion procedures. The inserted x-monotone curve is drawn with a light dashed line, surrounded by two solid arrows that represent the pair of twin half-edges added to the DCEL. Existing vertices are shown as black dots while new vertices are shown as light dots. Existing half-edges that are affected by the insertion operations are drawn as dashed arrows. (a) Inserting a curve as a new hole inside the face f. (b) Inserting a curve from an existing vertex u that corresponds to one of its endpoints. (c) Inserting an x-monotone curve whose endpoints are the already existing vertices u_1 and u_2. In our case, the new pair of half-edges close a new face f', where the hole h_1, which used to belong to f, now becomes an enclave in this new face.

```
  std::cout << arr.number_of_faces() << " faces:" << std::endl;
  for (fit = arr.faces_begin(); fit != arr.faces_end(); ++fit)
    print_face (fit);
}
```

32.2.3 Modifying the Arrangement

In this section we review the various member functions of the *Arrangement_2* class that allow users to modify the topological structure of the arrangement by introducing new edges or vertices, modifying them, or removing them.

The arrangement member-functions that insert new curves into the arrangement, thus enabling the construction of a planar subdivision, are rather specialized, as they require a-priori knowledge on the location of the inserted curve. Indeed, for most purposes it is more convenient to construct an arrangement using the free (global) insertion-functions.

Inserting Non-Intersecting x-Monotone Curves

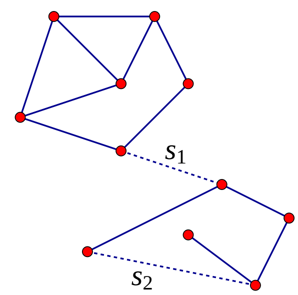
The most important functions that allow users to modify the arrangement, and perhaps the most frequently used ones, are the specialized insertion functions of x-monotone curves whose interior is disjoint from any other curve in the existing arrangement and do not contain any vertex of the arrangement. In addition, these function require that the location of the curve in the arrangement is known.

The motivation behind these rather harsh restrictions on the nature of the inserted curves is the decoupling of the topological arrangement representation from the various algorithms that operate on it. While the insertion of an x-monotone curve whose interior is disjoint from all existing arrangement features is quite straightforward (as we show next), inserting curves that intersect with the curves already inserted into the arrangement is much more complicated and requires the application of non-trivial geometric algorithms. These insertion operations are therefore implemented as free functions that operate on the arrangement and the inserted curve(s); see Section 32.4 for more details and examples.[5]

[5]You may skip to Section 32.4, and return to this subsection at a later point in time.

When an x-monotone curve is inserted into an existing arrangement, such that the interior of this curve is disjoint from any arrangement feature, only the following three scenarios are possible, depending on the status of the endpoints of the inserted subcurve:

1. In case both curve endpoints do not correspond to any existing arrangement vertex we have to create two new vertices corresponding to the curve endpoints and connect them using a pair of twin halfedges. This halfedge pair initiates a new hole inside the face that contains the curve in its interior.
2. If exactly one endpoint corresponds to an existing arrangement vertex (we distinguish between a vertex that corresponds to the left endpoint of the inserted curve and a vertex corresponding to its right endpoint), we have to create a new vertex that corresponds to the other endpoint of the curve and to connect the two vertices by a pair of twin halfedges that form an “antenna” emanating from the boundary of an existing connected component (note that if the existing vertex used to be isolated, this operation is actually equivalent to forming a new hole inside the face that contains this vertex).
3. If both endpoints correspond to existing arrangement vertices, we connect these vertices using a pair of twin halfedges. (If one or both vertices are isolated this case reduces to one of the two previous cases respectively.) The two following subcases may occur:
   - Two disconnected components are merged into a single connected component (as is the case with the segment s_1 in the figure to the left).
   - A new face is created, a face that splits from an existing arrangement face. In this case we also have to examine the holes and isolated vertices in the existing face and move the relevant ones inside the new face (as is the case with the segment s_2 in the figure to the left).



The *Arrangement_2* class offers insertion functions named *insert_in_face_interior()*, *insert_from_left_vertex()*, *insert_from_right_vertex()* and *insert_at_vertices()* that perform the special insertion procedures listed above. The first function accepts an x-monotone curve c and an arrangement face f that contains this curve in its interior. The other functions accept an x-monotone curve c and handles to the existing vertices that correspond to the curve endpoint(s). Each of the four functions returns a handle to one of the twin halfedges that have been created, where:

- *insert_in_face_interior(c, f)* returns a halfedge directed from the vertex corresponding to the left endpoint of c toward the vertex corresponding to its right endpoint.
- *insert_from_left_vertex(c, v)* and *insert_from_right_vertex(c, v)* returns a halfedge whose source is the vertex v that and whose target is the new vertex that has just been created.
- *insert_at_vertices(c, v1, v2)* returns a halfedge directed from v_1 to v_2.

The following program demonstrates the usage of the four insertion functions. It creates an arrangement of five line segments, as depicted in Figure 32.3.[6] As the arrangement is very simple, we use the simple Cartesian kernel of CGAL with integer coordinates for the segment endpoints. We also use the *Arr_segment_traits_2* class that enables the efficient maintenance of arrangements of line segments; see more details on this traits class in Section 32.6. This example, as many others in this chapter, uses some print-utility functions from the file *print_arr.h*; these functions are also listed in Section 32.2.2.

```
#include <CGAL/Simple_cartesian.h>
```

[6]Notice that in all figures in the rest of this chapter the coordinate axes are drawn only for illustrative purposes and are *not* part of the arrangement.

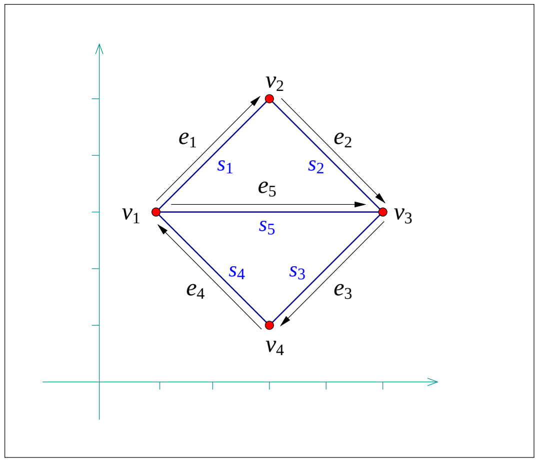


Figure 32.3: The arrangement of the line segments $s_1, \ldots, s_5$ constructed in *edge_insertion.cpp*. The arrows mark the direction of the halfedges returned from the various insertion functions.

```
#include <CGAL/Arr_segment_traits_2.h>
#include <CGAL/Arrangement_2.h>

#include "arr_print.h"

typedef int                                             Number_type;
typedef CGAL::Simple_cartesian<Number_type>             Kernel;
typedef CGAL::Arr_segment_traits_2<Kernel>              Traits_2;
typedef Traits_2::Point_2                               Point_2;
typedef Traits_2::X_monotone_curve_2                    Segment_2;
typedef CGAL::Arrangement_2<Traits_2>                   Arrangement_2;
typedef Arrangement_2::Vertex_handle                    Vertex_handle;
typedef Arrangement_2::Halfedge_handle                  Halfedge_handle;

int main()
{
  Arrangement_2   arr;

  Segment_2       s1(Point_2(1, 3), Point_2(3, 5));
  Segment_2       s2(Point_2(3, 5), Point_2(5, 3));
  Segment_2       s3(Point_2(5, 3), Point_2(3, 1));
  Segment_2       s4(Point_2(3, 1), Point_2(1, 3));
  Segment_2       s5(Point_2(1, 3), Point_2(5, 3));

  Halfedge_handle e1 = arr.insert_in_face_interior(s1,
   arr.unbounded_face());
  Vertex_handle   v1 = e1->source();
  Vertex_handle   v2 = e1->target();
  Halfedge_handle e2 = arr.insert_from_left_vertex(s2, v2);
  Vertex_handle   v3 = e2->target();
  Halfedge_handle e3 = arr.insert_from_right_vertex(s3, v3);
  Vertex_handle   v4 = e3->target();
  arr.insert_at_vertices(s4, v4, v1);
  arr.insert_at_vertices(s5, v1, v3);
```

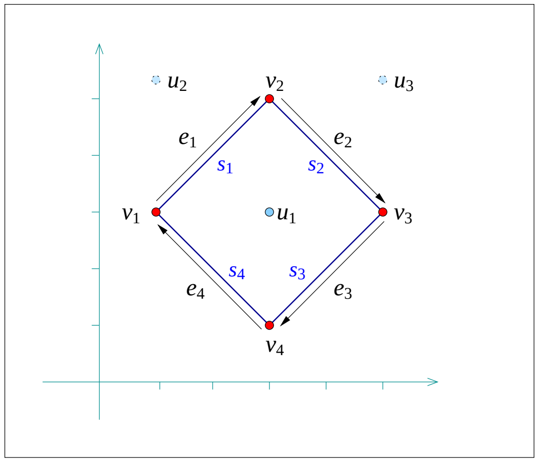


Figure 32.4: An arrangement of line segments containing three isolated vertices, as constructed in *isolated_vertices.cpp*. The vertices u_2 and u_3 are eventually removed from the arrangement.

```
  print_arrangement(arr);
  return 0;
}
```

File: examples/Arrangement_on_surface_2/edge_insertion.cpp

Observe that the first line segment is inserted in the interior of the unbounded face. The other line segments are inserted using the vertices created by the insertion of previous segments. The resulting arrangement consists of three faces, where the two bounded faces form together a hole in the unbounded face.

Manipulating Isolated Vertices

Isolated points are in general simpler geometric entities than curves and indeed the member functions that manipulate them are easier to understand.

The function *insert_in_face_interior(p, f)* inserts an isolated point p, located in the interior of a given face f, into the arrangement and returns a handle to the arrangement vertex it has created and associated with p. Naturally, this function has a precondition that p is really an isolated point, namely it does not coincide with any existing arrangement vertex and does not lie on any edge. As mentioned in Section 32.2.2, it is possible to obtain the face containing an isolated vertex handle v by calling *v->face()*.

The function *remove_isolated_vertex(v)* receives a handle to an isolated vertex and removes it from the arrangement.

The following program demonstrates the usage of the arrangement member-functions for manipulating isolated vertices. It first inserts three isolated vertices located inside the unbounded face, then it inserts four line segments that form a rectangular hole inside the unbounded face (see Figure 32.4 for an illustration). Finally, it traverses the vertices and removes those isolated vertices that are still contained in the unbounded face (u_2 and u_3 in this case):

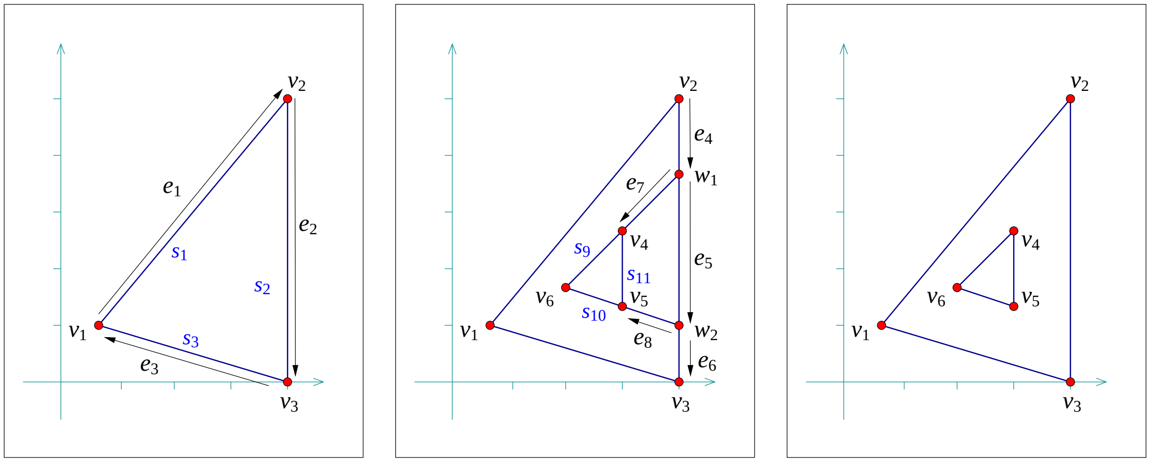
```
#include "arr_inexact_construction_segments.h"
#include "arr_print.h"

int main()
{
  // Insert isolated points.
  Arrangement arr;
  Face_handle uf = arr.unbounded_face();
  arr.insert_in_face_interior(Point(3, 3), uf);
  arr.insert_in_face_interior(Point(1, 5), uf);
  arr.insert_in_face_interior(Point(5, 5), uf);

  // Insert four segments that form a square-shaped face.
  Point p1(1, 3), p2(3, 5), p3(5, 3), p4(3, 1);
  Segment s1(p1, p2), s2(p2, p3), s3(p3, p4), s4(p4, p1);

  Halfedge_handle e1 = arr.insert_in_face_interior(s1, uf);
  Vertex_handle   v1 = e1->source();
  Vertex_handle   v2 = e1->target();
  Halfedge_handle e2 = arr.insert_from_left_vertex(s2, v2);
  Vertex_handle   v3 = e2->target();
  Halfedge_handle e3 = arr.insert_from_right_vertex(s3, v3);
  Vertex_handle   v4 = e3->target();
  arr.insert_at_vertices(s4, v4, v1);

  // Remove the isolated vertices located in the unbounded face.
  Arrangement::Vertex_iterator curr, next = arr.vertices_begin();
  for (curr = next++; curr != arr.vertices_end(); curr = next++) {
    // Keep an iterator to the next vertex, as curr might be deleted.
    if (curr->is_isolated() && curr->face() == uf)
      arr.remove_isolated_vertex(curr);
  }

  print_arrangement(arr);
  return 0;
}
```

File: examples/Arrangement_on_surface_2/isolated_vertices.cpp

Manipulating Halfedges

In the previous subsection we showed how to introduce new isolated vertices in the arrangement. But how does one create a vertex that lies on an existing arrangement edge (more precisely, on an x-monotone curve that is associated with an arrangement edge)?

It should be noted that such an operation involves the splitting of a curve at a given point in its interior, while the traits class used by *Arrangement_2* does not necessarily have the ability to perform such a split operation. However, if users have the ability to split an x-monotone curve into two at a given point p (this is usually the case when employing a more sophisticated traits class; see Section 32.6 for more details) they can use the *split_edge(e, c1, c2)* function, were c_1 and c_2 are the two subcurves resulting from splitting the x-monotone curve



(a) (b) (c)

Figure 32.5: An arrangement of line segments as constructed in *edge_manipulation.cpp*. Note that the edges e_7 and e_8 and the vertices w_1 and w_2, introduced in step (b) are eventually removed in step (c).

associated with the halfedge e at some point (call it p) in its interior. The function splits the halfedge pair into two pairs, both incident to a new vertex v associated with p, and returns a handle to a halfedge whose source equals e's source vertex and whose target is the new vertex v.

The reverse operation is also possible. Suppose that we have a vertex v of degree 2, whose two incident halfedges, e_1 and e_2, are associated with the curves c_1 and c_2. Suppose further that it is possible to merge these two curves into a single continuous x-monotone curve c. Calling *merge_edge(e1, e2, c)* will merge the two edges into a single edge associated with the curve c, essentially removing the vertex v from the arrangement.

Finally, the function *remove_edge(e)* removes the edge e from the arrangement. Note that this operation is the reverse of an insertion operation, so it may cause a connected component to split into two, or two faces to merge into one, or a hole to disappear. By default, if the removal of e causes one of its end-vertices to become isolated, we remove this vertex as well. However, users can control this behavior and choose to keep the isolated vertices by supplying additional Boolean flags to *remove_edge()* indicating whether the source and the target vertices are to be removed should they become isolated.

In the following example program we show how the edge-manipulation functions can be used. The program works in three steps, as demonstrated in Figure 32.5. Note that here we still stick to integer coordinates, but as we work on a larger scale we use an unbounded integer number-type (in this case, the *Gmpz* type taken from the GMP library) instead of the built-in *int* type.[7] In case the GMP library is not installed (as indicated by the *CGAL_USE_GMP* flag defined in *CGAL/basic.h*), we use *MP_Float*, a number-type included in CGAL's support library that is capable of storing floating-point numbers with unbounded mantissa. We also use the standard Cartesian kernel of CGAL as our kernel. This is recommended when the kernel is instantiated with a more complex number type, as we demonstrate in other examples in this chapter.

```
#include <CGAL/basic.h>

#include <CGAL/Exact_predicates_exact_constructions_kernel.h>
#include <CGAL/Arr_segment_traits_2.h>
#include <CGAL/Arrangement_2.h>
```

[7] As a rule of thumb, one can use a bounded integer type for representing line segments whose coordinates are bounded by $\lfloor \sqrt[3]{M} \rfloor$, where M is the maximal representable integer value. This guarantees that no overflows occur in the computations carried out by the traits class, hence all traits-class predicates always return correct results.

```
#include "arr_print.h"

typedef CGAL::Exact_predicates_exact_constructions_kernel   Kernel;
typedef CGAL::Arr_segment_traits_2<Kernel>                  Traits_2;
typedef Traits_2::Point_2                                   Point_2;
typedef Traits_2::X_monotone_curve_2                        Segment_2;
typedef CGAL::Arrangement_2<Traits_2>
   Arrangement_2;
typedef Arrangement_2::Vertex_handle
   Vertex_handle;
typedef Arrangement_2::Halfedge_handle
   Halfedge_handle;

int main()
{
  // Step(a) - construct a triangular face.
  Arrangement_2   arr;

  Segment_2       s1(Point_2(667, 1000), Point_2(4000, 5000));
  Segment_2       s2(Point_2(4000, 0), Point_2(4000, 5000));
  Segment_2       s3(Point_2(667, 1000), Point_2(4000, 0));

  Halfedge_handle e1 = arr.insert_in_face_interior(s1,
   arr.unbounded_face());
  Vertex_handle   v1 = e1->source();
  Vertex_handle   v2 = e1->target();
  Halfedge_handle e2 = arr.insert_from_right_vertex(s2, v2);
  Vertex_handle   v3 = e2->target();
  arr.insert_at_vertices(s3, v3, v1);

  // Step (b) - create additional two faces inside the triangle.
  Point_2         p1(4000, 3666), p2(4000, 1000);
  Segment_2       s4(Point_2(4000, 5000), p1);
  Segment_2       s5(p1, p2);
  Segment_2       s6(Point_2(4000, 0), p2);

  Halfedge_handle e4 = arr.split_edge(e2, s4, Segment_2(Point_2(4000, 0),
    p1));
  Vertex_handle   w1 = e4->target();
  Halfedge_handle e5 = arr.split_edge(e4->next(), s5, s6);
  Vertex_handle   w2 = e5->target();
  Halfedge_handle e6 = e5->next();

  Segment_2       s7(p1, Point_2(3000, 2666));
  Segment_2       s8(p2, Point_2(3000, 1333));
  Segment_2       s9(Point_2(3000, 2666), Point_2(2000, 1666));
  Segment_2       s10(Point_2(3000, 1333), Point_2(2000, 1666));
  Segment_2       s11(Point_2(3000, 1333), Point_2(3000, 2666));

  Halfedge_handle e7 = arr.insert_from_right_vertex(s7, w1);
  Vertex_handle   v4 = e7->target();
  Halfedge_handle e8 = arr.insert_from_right_vertex(s8, w2);
  Vertex_handle   v5 = e8->target();
```

```
  Vertex_handle   v6 =
    arr.insert_in_face_interior(Point_2(2000, 1666), e8->face());

  arr.insert_at_vertices(s9, v4, v6);
  arr.insert_at_vertices(s10, v5, v6);
  arr.insert_at_vertices(s11, v4, v5);

  // Step(c) - remove and merge faces to form a single hole in the
    traingle.
  arr.remove_edge(e7);
  arr.remove_edge(e8);

  e5 = arr.merge_edge(e5, e6, Segment_2(e5->source()->point(),
                                          e6->target()->point()));
  e2 = arr.merge_edge(e4, e5, s2);

  print_arrangement(arr);
  return 0;
}
```

File: examples/Arrangement_on_surface_2/edge_manipulation.cpp

Note how we use the halfedge handles returned from *split_edge()* and *merge_edge()*. Also note the insertion of the isolated vertex v_6 located inside the triangular face (the incident face of e_7). This vertex seizes from being isolated, as it is gets connected to other vertices.

In this context, we should mention the two member functions *modify_vertex(v, p)*, which sets p to be the point associated with the vertex v, and *modify_edge(e, c)*, which sets c to be the x-monotone curve associated with the halfedge e. These functions have preconditions that p is geometrically equivalent to *v->point()* and c is equivalent to *e->curve()* (i.e., the two curves have the same graph), respectively, to avoid the invalidation of the geometric structure of the arrangement. At a first glance it may seen as these two functions are of little use. However, we should keep in mind that there may be extraneous data (probably non-geometric) associated with the point objects or with the curve objects, as defined by the traits class. With these two functions we can modify this data; see more details in Section 32.6.

In addition, we can use these functions to replace a geometric object (a point or a curve) with an equivalent object that has a more compact representation. For example, we can replace the point $(\frac{20}{40}, \frac{99}{33})$ associated with some vertex v, by $(\frac{1}{2}, 3)$.

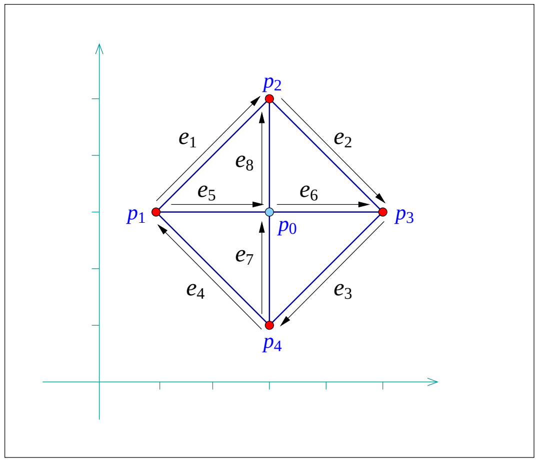
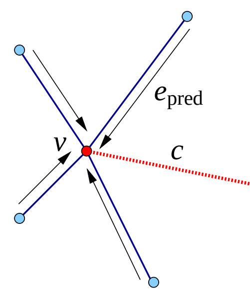


Figure 32.6: An arrangement of line segments, as constructed in *special_edge_insertion.cpp*. Note that p_0 is initially inserted as an isolated point and later on connected to the other four vertices.

advanced

Advanced Insertion Functions



Assume that the specialized insertion function *insert_from_left_vertex(c,v)* is invoked for a curve c, whose left endpoint is already associated with a non-isolated vertex v. Namely, v has already several incident halfedges. It is necessary in this case to locate the exact place for the new halfedge mapped to the inserted new curve c in the circular list of halfedges incident to v. More precisely, it is sufficient to locate one of the halfedges *pred* directed toward v such that c is located between *pred* and *pred->next()* in clockwise order around v, in order to complete the insertion (see Figure 32.2 for an illustration). This may take $O(d)$ time where d is the degree of the vertex. However, if the halfedge *pred* is known in advance, the insertion can be carried out in constant time.

The *Arrangement_2* class provides the advanced versions of the specialized insertion functions for a curve c — namely we have *insert_from_left_vertex(c,pred)* and *insert_from_right_vertex(c,pred)* that accept a halfedge *pred* as specified above, instead of a vertex v. These functions are more efficient, as they take constant time and do not perform any geometric operations. Thus, they should be used when the halfedge *pred* is known. In case that the vertex v is isolated or that the predecessor halfedge for the new inserted curve is not known, the simpler versions of these insertion functions should be used.

Similarly, there exist two overrides of the *insert_at_vertices()* function: One that accept the two predecessor halfedges around the two vertices v_1 and v_2 that correspond to the curve endpoints, and one that accepts a handle for one vertex and a predecessor halfedge around the other vertex.

The following program shows how to construct the arrangement depicted in Figure 32.6 using the specialized insertion functions that accept predecessor halfedges:

```
#include "arr_inexact_construction_segments.h"
#include "arr_print.h"

int main()
{
```

```
  Point           p0(3, 3), p1(1, 3), p2(3, 5), p3(5, 3), p4(3, 1);
  Segment         s1(p1, p2), s2(p2, p3), s3(p3, p4), s4(p4, p1);
  Segment         s5(p1, p0), s6(p0, p3), s7(p4, p0), s8(p0, p2);

  Arrangement     arr;
  Vertex_handle   v0 = arr.insert_in_face_interior(p0,
   arr.unbounded_face());
  Halfedge_handle e1 = arr.insert_in_face_interior(s1,
   arr.unbounded_face());
  Halfedge_handle e2 = arr.insert_from_left_vertex(s2, e1);
  Halfedge_handle e3 = arr.insert_from_right_vertex(s3, e2);
  Halfedge_handle e4 = arr.insert_at_vertices(s4, e3, e1->twin());
  Halfedge_handle e5 = arr.insert_at_vertices(s5, e1->twin(), v0);
  Halfedge_handle e6 = arr.insert_at_vertices(s6, e5, e3->twin());
  arr.insert_at_vertices(s7, e4->twin(), e6->twin());
  arr.insert_at_vertices(s8, e5, e2->twin());

  print_arrangement(arr);
  return 0;
}
```

File: examples/Arrangement_on_surface_2/special_edge_insertion.cpp

It is possible to perform even more refined operations on an *Arrangement_2* instance given specific topological information. As most of these operations are very fragile and perform no precondition testing on their input in order to gain efficiency, they are not included in the public interface of the arrangement class. Instead, the *Arr_accessor<Arrangement>* class allows access to these internal arrangement operations — see more details in the Reference Manual.

advanced

32.3 Issuing Queries on an Arrangement

One of the most important query types defined on arrangements is the *point-location* query: Given a point, find the arrangement cell that contains it. Typically, the result of a point-location query is one of the arrangement faces, but in degenerate situations the query point can be located on an edge or coincide with a vertex.

Point-location queries are not only common in many applications, they also play an important role in the free insertion-functions (see Section 32.4). Therefore, it is crucial to have the ability to answer such queries effectively for any arrangement instance.

32.3.1 Point-Location Queries

The arrangement package includes several classes (more precisely, class templates parameterized by an arrangement class) that model the *ArrangementPointLocation_2* concept. Namely, they all have a member function called *locate(q)* that accepts a query point q and result with a CGAL *Object* that wraps a handle to the arrangement cell containing the query point. This object can be assigned to either a *Face_const_handle*, *Halfedge_const_handle* or a *Vertex_const_handle*, depending on whether the query point is located inside a face, on an edge or on a vertex.

Note that the handles returned by the *locate()* functions are *const* (non-mutable) handles. If necessary, such handles may be casted to mutable handles using the static functions *Arrangement_on_surface_2::non_const_handle()* provided by the arrangement class.

An instance of any point-location class must be attached to an *Arrangement_on_surface_2* instance so we can use it to issue point-location queries. This attachment can be performed when the point-location instance is constructed, or at a later time, using the *init(arr)* method, where *arr* is the attached *Arrangement_on_surface_2* instance. In this chapter we always use the first option.

The following function template, which can be found in the example file *point_location_utils.h*, accepts a point-location object (whose type can be any of the models to the *ArrangementPointLocation_2* concept) and a query point, and prints out the result of the point-location query for the given point. Observe how we use the function *CGAL::assign()* is order to cast the resulting *CGAL::Object* into a handle to an arrangement feature. The point-location object *pl* is assumed to be already associated with an arrangement:

```
template <class PointLocation>
void point_location_query
        (const PointLocation& pl,
         const typename PointLocation::Arrangement_on_surface_2::Point_2& q)
{
  // Perform the point-location query.
  CGAL::Object obj = pl.locate (q);

  // Print the result.
  typedef typename PointLocation::Arrangement_on_surface_2  Arrangement_on_surface_2;

  typename Arrangement_on_surface_2::Vertex_const_handle    v;
  typename Arrangement_on_surface_2::Halfedge_const_handle  e;
  typename Arrangement_on_surface_2::Face_const_handle      f;

  std::cout << "The point " << q << " is located ";
  if (CGAL::assign (f, obj)) {
    // q is located inside a face:
    if (f->is_unbounded())
      std::cout << "inside the unbounded face." << std::endl;
    else
      std::cout << "inside a bounded face." << std::endl;
  }
  else if (CGAL::assign (e, obj)) {
    // q is located on an edge:
    std::cout << "on an edge: " << e->curve() << std::endl;
  }
  else if (CGAL::assign (v, obj)) {
    // q is located on a vertex:
    if (v->is_isolated())
      std::cout << "on an isolated vertex: " << v->point() << std::endl;
    else
      std::cout << "on a vertex: " << v->point() << std::endl;
  }
  else {
    CGAL_assertion_msg (false, "Invalid object.");
  }
}
```

Choosing a Point-Location Strategy

Each of the various point-location classes employs a different algorithm or *strategy*[8] for answering queries:

- *Arr_naive_point_location<Arrangement>* locates the query point naively, by exhaustively scanning all arrangement cells.
- *Arr_walk_along_a_line_point_location<Arrangement>* simulates a traversal, in reverse order, along an imaginary vertical ray emanating from the query point: It starts from the unbounded face of the arrangement and moves downward toward the query point until locating the arrangement cell containing it.
- *Arr_landmarks_point_location<Arrangement,Generator>* uses a set of "landmark" points whose location in the arrangement is known. Given a query point, it uses a KD-tree to find the nearest landmark and then traverses the straight line segment connecting this landmark to the query point.

  There are various ways to select the landmark set in the arrangement, determined by the *Generator* template parameter. By default, the generator class *Arr_landmarks_vertices_generator* is used and the arrangement vertices are the selected landmarks, but other landmark generators, such as sampling random points or choosing points on a grid, are also available; see the Reference Manual for more details.
- *Arr_trapezoid_ric_point_location<Arrangement>* implements a point location algorithm presented by Seidel [Sei91], which uses the randomized incremental construction described by Mulmuley [Mul90] (see also [dBvKOS00, Chapter 6]). The arrangement faces are decomposed into simpler cells of constant complexity known as *pseudo-trapezoids* and a search-structure (a directed acyclic graph) is constructed on top of these cells, allowing to locate the pseudo-trapezoid (hence the arrangement cell) containing a query point in logarithmic time.

The main advantage of the first two strategies is that they do not require any extra data, so the respective classes just store a pointer to an arrangement object and operate directly on it. Attaching such point-location objects to an existing arrangement has virtually no running-time cost at all, but the query time is linear in the size of the arrangement (the performance of the "walk" strategy is much better in practice, but its worst-case performance is linear). Using these strategies is therefore recommended only when a relatively small number of point-location queries are issued by the application, or when the arrangement is constantly changing (i.e., changes in the arrangement structure are more frequent than point-location queries).

On the other hand, the landmarks and the trapezoid RIC strategies require auxiliary data structures on top of the arrangement, which they need to construct once they are attached to an arrangement object and need to keep up-to-date as this arrangement changes. The data structure needed by the landmarks strategy can be constructed in $O(N \log N)$ time (where N is the overall number of edges in the arrangement), whereas the construction of the data structure needed by the trapezoid RIC strategy takes expected $O(N \log N)$, but may require several rebuilds. As shown in [HKH12], one can expect only a small number of rebuilds. In practice, the construction needed by the landmark algorithm is significantly faster. In addition, although both resulting data structures are asymptotically linear in size, the KD-tree that the landmark algorithm stores needs significantly less memory.

We note that the trapezoid RIC algorithm guarantees a logarithmic query time for any scenario making it advantageous comparing to other strategies. The query time of the landmarks strategy, on the other hand, is only logarithmic on average — and we may have scenarios where the query time can be linear. In practice, in many scenarios the query times of both strategies are competitive. For a detailed experimental comparison, see [HH08].

Another advantageous feature of the trapezoid RIC strategy when compared with other methods is the unbounded curves support. Currently, this is the most efficient strategy suitable for general unbounded subdivisions.

[8] We use the term *strategy* following the design pattern taxonomy [GHJV95].

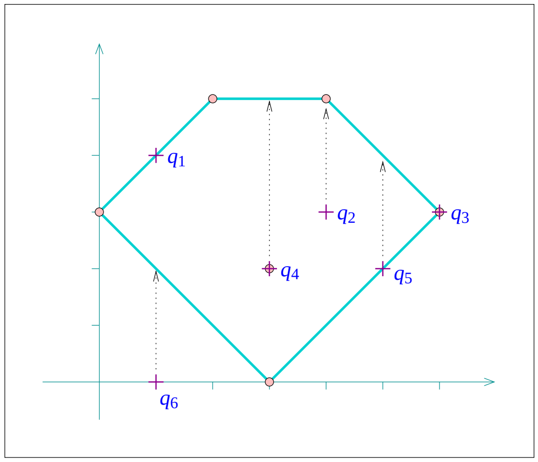


Figure 32.7: The arrangement of line segments, as constructed in *point_location.cpp*, *vertical_ray_shooting.cpp*, and *batched_point_location.cpp*. The arrangement vertices are drawn as small discs, while the query points $q_1, \ldots, q_6$ are marked with crosses.

The main drawback in the current implementation of the landmark strategy, compared to the trapezoidal RIC strategy, is that while the updating the auxiliary data structures related to the trapezoidal decomposition is done very efficiently, the KD-tree maintained by the landmark algorithm needs to be frequently rebuilt as the arrangement changes. In addition, using the landmark point-location class adds some extra requirement from the traits class (that is, the traits class should be a model of a refined concept *ArrangementLandmarkTraits_2*; see Section 32.6 for the details). However, most built-in traits classes that come with the CGAL public release support these extra operations.

It is therefore recommended to use the *Arr_landmarks_point_location* class when the application frequently issues point-location queries on a bounded arrangement that only seldom changes. If the arrangement consists of unbounded curves or is more dynamic and is frequently going through changes, the *Arr_trapezoid_ric_point_location* class should be the selected point-location strategy.

An Example

The following program constructs a simple arrangement of five line segments that form a pentagonal face, with a single isolated vertex in its interior, as depicted in Figure 32.7 (the arrangement construction is performed by the function *construct_segment_arr()* whose listing is omitted here and can be found in *point_location_utils.h*). It then employs the naive and the landmark strategies to issue several point-location queries on this arrangement:

```
#include <CGAL/Simple_cartesian.h>
#include <CGAL/Arr_segment_traits_2.h>
#include <CGAL/Arrangement_2.h>
#include <CGAL/Arr_naive_point_location.h>
#include <CGAL/Arr_landmarks_point_location.h>

#include "point_location_utils.h"

typedef int
   Number_type;
typedef CGAL::Simple_cartesian<Number_type>                   Kernel;
typedef CGAL::Arr_segment_traits_2<Kernel>                    Traits_2;
```

```
typedef Traits_2::Point_2                                       Point_2;
typedef CGAL::Arrangement_2<Traits_2>
    Arrangement_2;
typedef CGAL::Arr_naive_point_location<Arrangement_2>           Naive_pl;
typedef CGAL::Arr_landmarks_point_location<Arrangement_2>
    Landmarks_pl;

int main ()
{
  // Construct the arrangement.
  Arrangement_2    arr;
  Naive_pl         naive_pl (arr);
  Landmarks_pl     landmarks_pl;

  construct_segments_arr (arr);

  // Perform some point-location queries using the naive strategy.
  Point_2          q1 (1, 4);
  Point_2          q2 (4, 3);
  Point_2          q3 (6, 3);

  point_location_query (naive_pl, q1);
  point_location_query (naive_pl, q2);
  point_location_query (naive_pl, q3);

  // Attach the landmarks object to the arrangement and perform queries.
  Point_2          q4 (3, 2);
  Point_2          q5 (5, 2);
  Point_2          q6 (1, 0);

  landmarks_pl.attach (arr);

  point_location_query (landmarks_pl, q4);
  point_location_query (landmarks_pl, q5);
  point_location_query (landmarks_pl, q6);

  return 0;
}
```

File: examples/Arrangement_on_surface_2/point_location.cpp

Note that the program uses the auxiliary *point_location_query()* function template to nicely print the result of each query. This function can be found in the header file *point_location_utils.h*.

32.3.2 Vertical Ray Shooting

Another important query issued on arrangements is the vertical ray-shooting query: Given a query point, which arrangement feature do we encounter if we shoot a vertical ray emanating upward (or downward) from this point? In the general case the ray hits an edge, but it is possible that it hits a vertex, or that the arrangement does not have any feature lying directly above (or below) the query point.

All point-location classes listed in the previous section are also models of the *ArrangementVerticalRayShoot_2* concept. That is, they all have member functions called *ray_shoot_up(q)* and *ray_shoot_down(q)* that accept a query point *q* and output a CGAL *Object*. This can be assigned to either a *Halfedge_const_handle* or to a *Vertex_const_handle*. Alternatively, the returned value is a *Face_const_handle* for the unbounded face of the arrangement, in case there is no edge or vertex lying directly above (or below) *q*.

The following function template, *vertical_ray_shooting_query()*, which also located in the header file *point_location_utils.h*, accepts a vertical ray-shooting object, whose type can be any of the models to the *ArrangementVerticalRayShoot_2* concept and prints out the result of the upward vertical ray-shooting operations from a given query point. The ray-shooting object *vrs* is assumed to be already associated with an arrangement:

```
template <class VerticalRayShoot>
void vertical_ray_shooting_query
    (const VerticalRayShoot& vrs,
     const typename VerticalRayShoot::Arrangement_on_surface_2::Point_2& q)
{
  // Perform the point-location query.
  CGAL::Object    obj = vrs.ray_shoot_up (q);

  // Print the result.
  typedef typename VerticalRayShoot::Arrangement_on_surface_2  Arrangement_on_surface_2;

  typename Arrangement_on_surface_2::Vertex_const_handle    v;
  typename Arrangement_on_surface_2::Halfedge_const_handle  e;
  typename Arrangement_on_surface_2::Face_const_handle      f;

  std::cout << "Shooting up from " << q << " : ";
  if (CGAL::assign (e, obj)) {
    // We hit an edge:
    std::cout << "hit an edge: " << e->curve() << std::endl;
  }
  else if (CGAL::assign (v, obj)) {
    // We hit a vertex:
    if (v->is_isolated())
      std::cout << "hit an isolated vertex: " << v->point() << std::endl;
    else
      std::cout << "hit a vertex: " << v->point() << std::endl;
  }
  else if (CGAL::assign (f, obj)) {
    // We did not hit anything:
    CGAL_assertion (f->is_unbounded());

    std::cout << "hit nothing." << std::endl;
  }
  else {
    CGAL_assertion_msg (false, "Invalid object.");
  }
}
```

The following program uses the auxiliary function listed above to perform vertical ray-shooting queries on an arrangement. The arrangement and the query points are exactly the same as in *point_location.cpp* (see Figure 32.7):

```
#include <CGAL/Cartesian.h>
#include <CGAL/MP_Float.h>
#include <CGAL/Arr_segment_traits_2.h>
#include <CGAL/Arrangement_2.h>
#include <CGAL/Arr_walk_along_line_point_location.h>
#include <CGAL/Arr_trapezoid_ric_point_location.h>

#include "point_location_utils.h"

typedef CGAL::MP_Float
   Number_type;
typedef CGAL::Cartesian<Number_type>                             Kernel;
typedef CGAL::Arr_segment_traits_2<Kernel>                       Traits_2;
typedef Traits_2::Point_2                                        Point_2;
typedef CGAL::Arrangement_2<Traits_2>
   Arrangement_2;
typedef CGAL::Arr_walk_along_line_point_location<Arrangement_2> Walk_pl;
typedef CGAL::Arr_trapezoid_ric_point_location<Arrangement_2>   Trap_pl;

int main ()
{
  // Construct the arrangement.
  Arrangement_2     arr;
  Walk_pl           walk_pl (arr);
  Trap_pl           trap_pl;

  construct_segments_arr (arr);

  // Perform some vertical ray-shooting queries using the walk strategy.
  Point_2           q1 (1, 4);
  Point_2           q2 (4, 3);
  Point_2           q3 (6, 3);

  vertical_ray_shooting_query (walk_pl, q1);
  vertical_ray_shooting_query (walk_pl, q2);
  vertical_ray_shooting_query (walk_pl, q3);

  // Attach the trapezoid-RIC object to the arrangement and perform
   queries.
  Point_2           q4 (3, 2);
  Point_2           q5 (5, 2);
  Point_2           q6 (1, 0);

  trap_pl.attach (arr);
  vertical_ray_shooting_query (trap_pl, q4);
  vertical_ray_shooting_query (trap_pl, q5);
  vertical_ray_shooting_query (trap_pl, q6);

  return 0;
}
```

File: examples/Arrangement_on_surface_2/vertical_ray_shooting.cpp

The number type we use in this example is CGAL's built-in *MP_Float* type which is a floating-point number with an unbounded mantissa and a 32-bit exponent. It supports construction from an integer or from a machine *float* or *double* and performs additions, subtractions and multiplications in an exact number.

32.3.3 Batched Point-Location

Suppose that at a given moment our application has to issue a relatively large number m of point-location queries on a specific arrangement instance. It is possible of course to define a point-location object and to issue separate queries on the arrangement. However, as explained in Section 32.3.1, choosing a simple point-location strategy (either the naive or the walk strategy) means inefficient queries, while the more sophisticated strategies need to construct auxiliary structures that incur considerable overhead in running time.

On the other hand, the arrangement package includes a free *locate()* function that accepts an arrangement a range of query points as its input and sweeps through the arrangement to locate all query points in one pass. The function outputs the query results as pairs, where each pair is comprised of a query point and a CGAL *Object* that represents the cell containing the point (see Section 32.3.1). The output pairs are sorted in increasing xy-lexicographical order of the query point.

The batched point-location operation can be performed in $O((m+N)\log(m+N))$ time, where N is the number of edges in the arrangement. This means that when the number m of point-location queries is of the same order of magnitude as N, this operation is more efficient than issuing separate queries. This suggestion is also backed up by experimental results. Moreover, the batched point-location operation is also advantageous as it does not have to construct and maintain additional data structures.

The following program issues a batched point-location query, which is essentially equivalent to the six separate queries performed in *point_location.cpp* (see Section 32.3.1):

```
#include <CGAL/Cartesian.h>
#include <CGAL/MP_Float.h>
#include <CGAL/Arr_segment_traits_2.h>
#include <CGAL/Arrangement_2.h>
#include <CGAL/Arr_batched_point_location.h>
#include <list>

#include "point_location_utils.h"

typedef CGAL::MP_Float                                     Number_type;
typedef CGAL::Cartesian<Number_type>                       Kernel;
typedef CGAL::Arr_segment_traits_2<Kernel>                 Traits_2;
typedef Traits_2::Point_2                                  Point_2;
typedef CGAL::Arrangement_2<Traits_2>                      Arrangement_2;
typedef std::pair<Point_2, CGAL::Object>                   Query_result;

int main ()
{
  // Construct the arrangement.
  Arrangement_2    arr;

  construct_segments_arr (arr);

  // Perform a batched point-location query.
  std::list<Point_2>       query_points;
```

```
  std::list<Query_result>  results;

  query_points.push_back (Point_2 (1, 4));
  query_points.push_back (Point_2 (4, 3));
  query_points.push_back (Point_2 (6, 3));
  query_points.push_back (Point_2 (3, 2));
  query_points.push_back (Point_2 (5, 2));
  query_points.push_back (Point_2 (1, 0));

  locate (arr, query_points.begin(), query_points.end(),
          std::back_inserter (results));

  // Print the results.
  std::list<Query_result>::const_iterator  res_iter;
  Arrangement_2::Vertex_const_handle       v;
  Arrangement_2::Halfedge_const_handle     e;
  Arrangement_2::Face_const_handle         f;

  for (res_iter = results.begin(); res_iter != results.end();
   ++res_iter) {
    std::cout << "The point (" << res_iter->first << ") is located ";
    if (CGAL::assign (f, res_iter->second)) {
      // The current qeury point is located inside a face:
      if (f->is_unbounded())
        std::cout << "inside the unbounded face." << std::endl;
      else
        std::cout << "inside a bounded face." << std::endl;
    }
    else if (CGAL::assign (e, res_iter->second)) {
      // The current qeury point is located on an edge:
      std::cout << "on an edge: " << e->curve() << std::endl;
    }
    else if (CGAL::assign (v, res_iter->second)) {
      // The current qeury point is located on a vertex:
      if (v->is_isolated())
        std::cout << "on an isolated vertex: " << v->point() <<
   std::endl;
      else
        std::cout << "on a vertex: " << v->point() << std::endl;
    }
  }

  return 0;
}
```

File: examples/Arrangement_on_surface_2/batched_point_location.cpp

32.4 Free Functions in the Arrangement Package

In Section 32.2 we reviewed in details the *Arrangement_2* class, which represents two-dimensional subdivisions induced by planar curves, and mentioned that its interface is minimal in the sense that the member functions hardly perform any geometric algorithms and are mainly used for maintaining the topological structure of the subdivision. In this section we explain how to utilize the free (global) functions that operate on arrangements. The implementation of these operations typically require non-trivial geometric algorithms or load some extra requirements on the traits class.

32.4.1 Incremental Insertion Functions

Inserting Non-Intersecting Curves

In Section 32.2 we explained how to construct arrangements of x-monotone curves that are pairwise disjoint in their interior, when the location of the segment endpoints in the arrangement is known. Here we relax this constraint, and allow the location of the inserted x-monotone curve endpoints to be arbitrary, as it may be unknown at the time of insertion. We retain, for the moment, the requirement that the interior of the inserted curve is disjoint from all existing arrangement edges and vertices.

The free function *insert_non_intersecting_curve(arr, c, pl)* inserts the x-monotone curve c into the arrangement *arr*, with the precondition that the interior of c is disjoint from all *arr*'s existing edges and vertices. The third argument *pl* is a point-location object attached to the arrangement, which is used for performing the insertion. It locates both curve endpoints in the arrangement, where each endpoint is expected to either coincide with an existing vertex or lie inside a face. It is possible to invoke one of the specialized insertion functions (see Section 32.2), based on the query results, and insert c at its proper position.[9] The insertion operation thus hardly requires any geometric operations on top on the ones needed to answer the point-location queries. Moreover, it is sufficient that the arrangement class is instantiated with a traits class that models the *ArrangementBasicTraits_2* concept (or the *ArrangementLandmarkTraits_2* concept, if the landmark point-location strategy is used), which does not have to support the computation of intersection points between curves.

The variant *insert_non_intersecting_curve(arr, c)* is also available. Instead of accepting a user-defined point-location object, it defines a local instance of the walk point-location class and uses it to insert the curve.

Inserting x-Monotone Curves

The *insert_non_intersecting_curve()* function is very efficient, but its preconditions on the input curves are still rather restricting. Let us assume that the arrangement is instantiated with a traits class that models the refined *ArrangementXMonotoneTraits_2* concept and supports intersection computations (see Section 32.6 for the exact details). Given an x-monotone curve, it is sufficient to locate its left endpoint in the arrangement and to trace its *zone*, namely the set of arrangement features crossing the curve, until the right endpoint is reached. Each time the new curve c crosses an existing vertex or an edge, the curve is split into subcurves (in the latter case, we have to split the curve associated with the existing halfedge as well) and associate new edges with the resulting subcurves. Recall that an edge is represented by a pair of twin halfedges, so we split it into two halfedge pairs.

The free function *insert(arr, c, pl)* performs this insertion operation. It accepts an x-monotone curve c, which may intersect some of the curves already in the arrangement *arr*, and inserts it into the arrangement by computing its zone. Users may supply a point-location object *pl*, or use the default walk point-location strategy (namely, the variant *insert(arr, c)* is also available). The running-time of this insertion function is proportional to the complexity of the zone of the curve c.

advanced

In some cases users may have a prior knowledge of the location of the left endpoint of the x-monotone curve c they wish to insert, so they can perform the insertion without issuing any point-location queries. This can

[9]The *insert_non_intersecting_curve()* function, as all other functions reviewed in this section, is a function template, parameterized by an arrangement class and a point-location class (a model of the *ArrangementPointLocation_2* concept).

be done by calling *insert(arr, c, obj)*, where *obj* is an object represents the location of *c*'s left endpoint in the arrangement — namely it wraps a *Vertex_const_handle*, a *Halfedge_const_handle* or a *Face_const_handle* (see also Section 32.3.1).

advanced

Inserting General Curves

So far all our examples were of arrangements of line segments, where the *Arrangement_2* template was instantiated with the *Arr_segment_traits_2* class. In this case, the fact that *insert()* accepts an x-monotone curve does not seem to be a restriction, as all line segments are x-monotone (note that we consider vertical line segments to be *weakly* x-monotone).

Suppose that we construct an arrangement of circles. A circle is obviously not x-monotone, so we cannot insert it in the same way we inserted x-monotone curves. [10] However, it is possible to subdivide each circle into two x-monotone circular arcs (its upper half and its lower half) and to insert each x-monotone arc separately.

The free function *insert()* also supports general curve and not necessarily x-monotone curves. In this case it requires that the traits class used by the arrangement *arr* to be a model of the concept *ArrangementTraits_2*, which refines the *ArrangementXMonotoneTraits_2* concept. It has to define an additional *Curve_2* type (which may differ from the *X_monotone_curve_2* type), and support the subdivision of curves of this new type into x-monotone curves (see the exact details in Section 32.6). The *insert(arr, c, pl)* function performs the insertion of the curve c, which does not need to be x-monotone, into the arrangement by subdividing it (if needed) into x-monotone subcurves and inserting each one separately. Users may supply a point-location object *pl*, or use the default walk point-location strategy by calling *insert(arr, c)*.

Inserting Points

The arrangement class enables us to insert a point as an isolated vertex in a given face. The free function *insert_point(arr, p, pl)* inserts a vertex into *arr* that corresponds to the point p at an arbitrary location. It uses the point-location object *pl* to locate the point in the arrangement (by default, the walk point-location strategy is used), and acts according to the result as follows:

- If p is located inside a face, it is inserted as an isolated vertex inside this face.
- If p lies on an edge, the edge is split to create a vertex associated with p.
- Otherwise, p coincides with an existing vertex and we are done.

In all cases, the function returns a handle to the vertex associated with p.

The arrangement *arr* should be instantiated with a traits class that models the *ArrangementXMonotoneTraits_2* concept, as the insertion operation may involve splitting curves.

An Example

The program below constructs an arrangement of intersecting line-segments. We know that s_1 and s_2 do not intersect, so we use *insert_non_intersecting_curve()* to insert them into the empty arrangement. The rest of the

[10] Note that a key operation performed by *insert()* is to locate the left endpoint of the curve in the arrangement. A circle, however, does not have any endpoints!

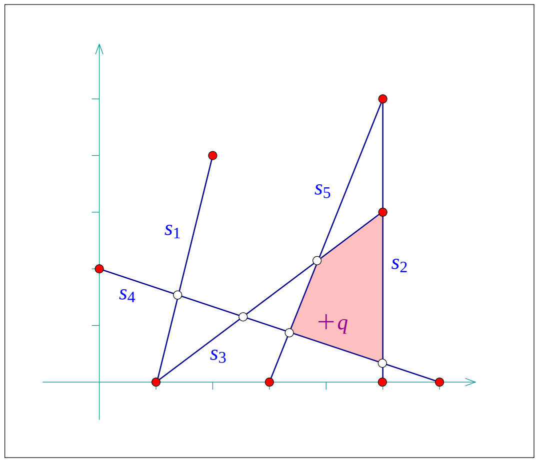


Figure 32.8: An arrangement of five intersecting line segments, as constructed in *incremental_insertion.cpp* and *aggregated_insertion.cpp*. The segment endpoints are marked by black disks and the arrangement vertices that correspond to intersection points are marked by circles. The query point q is marked with a cross and the face that contains it is shaded.

segments are inserted using *insert()*. The resulting arrangement consists of 13 vertices, 16 edges, and 5 faces, as can be seen in Figure 32.8.

In the earlier examples, all arrangement vertices corresponded to segment endpoints. In this example we have additional vertices that correspond to intersection points between two segments. The coordinates of these intersection points are rational numbers, if the input coordinates are rational (or integer). Therefore, the *Quotient*< *int*> number type is used to represent the coordinates:

```
#include <CGAL/Cartesian.h>
#include <CGAL/Quotient.h>
#include <CGAL/Arr_segment_traits_2.h>
#include <CGAL/Arrangement_2.h>
#include <CGAL/Arr_walk_along_line_point_location.h>

#include "arr_print.h"

typedef CGAL::Quotient<int>
    Number_type;
typedef CGAL::Cartesian<Number_type>                            Kernel;
typedef CGAL::Arr_segment_traits_2<Kernel>                      Traits_2;
typedef Traits_2::Point_2                                       Point_2;
typedef Traits_2::X_monotone_curve_2
    Segment_2;
typedef CGAL::Arrangement_2<Traits_2>
    Arrangement_2;
typedef CGAL::Arr_walk_along_line_point_location<Arrangement_2> Walk_pl;

int main ()
{
  // Construct the arrangement of five intersecting segments.
  Arrangement_2  arr;
  Walk_pl        pl(arr);
```

```
  Segment_2      s1 (Point_2(1, 0), Point_2(2, 4));
  Segment_2      s2 (Point_2(5, 0), Point_2(5, 5));
  Segment_2      s3 (Point_2(1, 0), Point_2(5, 3));
  Segment_2      s4 (Point_2(0, 2), Point_2(6, 0));
  Segment_2      s5 (Point_2(3, 0), Point_2(5, 5));

  insert_non_intersecting_curve (arr, s1, pl);
  insert_non_intersecting_curve (arr, s2, pl);
  insert (arr, s3, pl);
  insert (arr, s4, pl);
  insert (arr, s5, pl);

  // Print the size of the arrangement.
  std::cout << "The arrangement size:" << std::endl
            << "   V = " << arr.number_of_vertices()
            << ",  E = " << arr.number_of_edges()
            << ",  F = " << arr.number_of_faces() << std::endl;

  // Perform a point-location query on the resulting arrangement and
    print
  // the boundary of the face that contains it.
  Point_2        q (4, 1);
  CGAL::Object   obj = pl.locate (q);

  Arrangement_2::Face_const_handle  f;
  CGAL_assertion_code (bool           success =) CGAL::assign (f, obj);

  CGAL_assertion (success);
  std::cout << "The query point (" << q << ") is located in: ";
  print_face<Arrangement_2> (f);

  return 0;
}
```

File: examples/Arrangement_on_surface_2/incremental_insertion.cpp

Other Zone Related Functions

In this section we have described so far free functions that insert curves and points to a given arrangement. Now we will describe functions that don't insert curves or points to an arrangement nor do they change the arrangement, but they are closely related to the incremental insertion functions as they also use the zone algorithm.

The free function *do_intersect()* checks if a given curve or x-monotone curve intersects an existing arrangement's edges or vertices. If the give curve is not an x-monotone curve then the function subdivides the given curve into x-monotone subcurves and isolated vertices . Each subcurve is in turn checked for intersection. The function uses the zone algorithm to check if the curve intersects the arrangement. First, the curve's left endpoint is located. Then, its zone is computed starting from its left endpoint location. The zone computation terminates when an intersection with an arrangement's edge/vertex is found or when the right endpoint is reached. A given point-location object is used for locating the left endpoint of the given curve in the existing arrangement. By default, the function uses the "walk along line" point-location strategy — namely an instance of the class *Arr_walk_along_line_point_location*. If the given curve is x-monotone then the traits class must model the

ArrangementXMonotoneTraits_2 concept. If the curve is not x-monotone curve then the traits class must model the *ArrangementTraits_2* concept.

The *zone()* function computes the zone of a given x-monotone curve in a given arrangement. Meaning, it outputs all the arrangement's elements (vertices, edges and faces) that the x-monotone curve intersects in the order that they are discovered when traversing the x-monotone curve from left to right. The function uses a given point-location object to locate the left endpoint of the given x-monotone curve. By default, the function uses the "walk along line" point-location strategy. The function requires that the traits class will model the *ArrangementXMonotoneTraits_2* concept.

32.4.2 Aggregated Insertion Functions

Let us assume that we have to insert a set of m input curves into an arrangement. It is possible to do this incrementally, inserting the curves one by one, as shown in the previous section. However, the arrangement package provides three free functions that aggregately insert a range of curves into an arrangement:

- *insert_non_intersecting_curves(arr, begin, end)* inserts a range of x-monotone curves given by the input iterators *[begin, end)* into an arrangement *arr*. The x-monotone curves should be pairwise disjoint in their interior and also interior-disjoint from all existing edges and vertices of *arr*.
- *insert(arr, begin, end)* inserts a range of general (not necessarily x-monotone) curves of type *Curve_2* or *X_monotone_curve_2* that may intersect one another, given by the input iterators *[begin, end)*, into the arrangement *arr*.

We distinguish between two cases: (i) The given arrangement *arr* is empty (has only an unbounded face), so we have to construct it from scratch. (ii) We have to insert m input curves to a non-empty arrangement *arr*.

In the first case, we sweep over the input curves, compute their intersection points and construct the DCEL that represents their planar arrangement. This process is performed in $O\left((m+k)\log m\right)$ time, where k is the total number of intersection points. The running time is asymptotically better than the time needed for incremental insertion, if the arrangement is relatively sparse (when k is bounded by $\frac{m^2}{\log m}$), but in practice it is recommended to use this aggregated construction process even for dense arrangements, since the sweep-line algorithm needs less geometric operations compared to the incremental insertion algorithms and hence typically runs much faster in practice.

Another important advantage the aggregated insertion functions have is that they do not issue point-location queries. Thus, no point-location object needs to be attached to the arrangement. As explained in Section 32.3.1, there is a trade-off between construction time and query time in each of the point-location strategies, which affects the running times of the incremental insertion process. Naturally, this trade-off is irrelevant in case of aggregated insertion as above.

The example below shows how to construct the arrangement of line segments depicted in Figure 32.8 and built incrementally in *incremental_insertion.cpp*, as shown in the previous section. We use the aggregated insertion function *insert()* as we deal with line segments. Note that no point-location object needs to be defined and attached to the arrangement:

```
#include <CGAL/Cartesian.h>
#include <CGAL/Quotient.h>
#include <CGAL/Arr_segment_traits_2.h>
#include <CGAL/Arrangement_2.h>
#include <list>
```

```
typedef CGAL::Quotient<int>                              Number_type;
typedef CGAL::Cartesian<Number_type>                     Kernel;
typedef CGAL::Arr_segment_traits_2<Kernel>               Traits_2;
typedef Traits_2::Point_2                                Point_2;
typedef Traits_2::X_monotone_curve_2                     Segment_2;
typedef CGAL::Arrangement_2<Traits_2>                    Arrangement_2;

int main ()
{
  // Construct the arrangement of five intersecting segments.
  Arrangement_2            arr;
  std::list<Segment_2>     segments;

  segments.push_back (Segment_2 (Point_2(1, 0), Point_2(2, 4)));
  segments.push_back (Segment_2 (Point_2(5, 0), Point_2(5, 5)));
  segments.push_back (Segment_2 (Point_2(1, 0), Point_2(5, 3)));
  segments.push_back (Segment_2 (Point_2(0, 2), Point_2(6, 0)));
  segments.push_back (Segment_2 (Point_2(3, 0), Point_2(5, 5)));

  insert (arr, segments.begin(), segments.end());

  // Print the size of the arrangement.
  std::cout << "The arrangement size:" << std::endl
            << "   V = " << arr.number_of_vertices()
            << ",  E = " << arr.number_of_edges()
            << ",  F = " << arr.number_of_faces() << std::endl;

  return 0;
}
```

File: examples/Arrangement_on_surface_2/aggregated_insertion.cpp

In case we have to insert a set of m curves into an existing arrangement, where we denote the number of edges in the arrangement by N. As a rule of thumb, if $m = o(\sqrt{N})$, we insert the curves one by one. For larger input sets, we use the aggregated insertion procedures.

In the example below we aggregately construct an arrangement of a set $\mathcal{S}_1$ containing five line segments. Then we insert a single segment using the incremental insertion function. Finally, we add a set $\mathcal{S}_2$ with five more line segments in an aggregated fashion. Notice that the line segments of $\mathcal{S}_1$ are pairwise interior-disjoint, so we use *insert_non_intersecting_curves()*. $\mathcal{S}_2$ also contain pairwise interior-disjoint segments, but as they intersect the existing arrangement, we have to use *insert()* to insert them. Also note that the single segment s we insert incrementally overlaps an existing arrangement edge:

```
#include <CGAL/Cartesian.h>
#include <CGAL/Quotient.h>
#include <CGAL/MP_Float.h>
#include <CGAL/Arr_segment_traits_2.h>
#include <CGAL/Arrangement_2.h>
#include <CGAL/Arr_naive_point_location.h>

#include "arr_print.h"
```

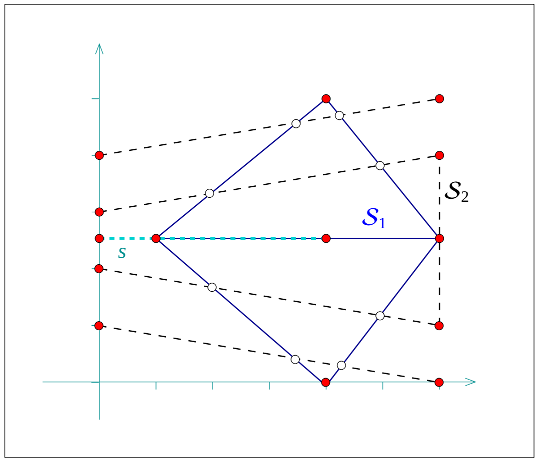


Figure 32.9: An arrangement of intersecting line segments, as constructed in *global_insertion.cpp*. The segments of $\mathcal{S}_1$ are drawn in solid lines and the segments of $\mathcal{S}_2$ are drawn in dark dashed lines. Note that the segment s (light dashed line) overlaps one of the segments in $\mathcal{S}_1$.

```
typedef CGAL::Quotient<CGAL::MP_Float>                   Number_type;
typedef CGAL::Cartesian<Number_type>                     Kernel;
typedef CGAL::Arr_segment_traits_2<Kernel>               Traits_2;
typedef Traits_2::Point_2                                Point_2;
typedef Traits_2::X_monotone_curve_2                     Segment_2;
typedef CGAL::Arrangement_2<Traits_2>                    Arrangement_2;
typedef CGAL::Arr_naive_point_location<Arrangement_2> Naive_pl;

int main ()
{
  // Construct the arrangement of five intersecting segments.
  Arrangement_2      arr;
  Segment_2          S1 [5];

  S1[0] = Segment_2 (Point_2 (1, 2.5), Point_2 (4, 5));
  S1[1] = Segment_2 (Point_2 (1, 2.5), Point_2 (6, 2.5));
  S1[2] = Segment_2 (Point_2 (1, 2.5), Point_2 (4, 0));
  S1[3] = Segment_2 (Point_2 (4, 5), Point_2 (6, 2.5));
  S1[4] = Segment_2 (Point_2 (4, 0), Point_2 (6, 2.5));

  insert_non_intersecting_curves (arr, S1, S1 + 5);

  // Perform an incremental insertion of a single overlapping segment.
  Naive_pl           pl (arr);

  insert (arr, Segment_2 (Point_2 (0, 2.5), Point_2 (4, 2.5)), pl);

  // Aggregately insert an additional set of five segments.
  Segment_2          S2 [5];

  S2[0] = Segment_2 (Point_2 (0, 4), Point_2 (6, 5));
  S2[1] = Segment_2 (Point_2 (0, 3), Point_2 (6, 4));
  S2[2] = Segment_2 (Point_2 (0, 2), Point_2 (6, 1));
```

```
  S2[3] = Segment_2 (Point_2 (0, 1), Point_2 (6, 0));
  S2[4] = Segment_2 (Point_2 (6, 1), Point_2 (6, 4));

  insert (arr, S2, S2 + 5);

  // Print the size of the arrangement.
  std::cout << "The arrangement size:" << std::endl
            << "   V = " << arr.number_of_vertices()
            << ",  E = " << arr.number_of_edges()
            << ",  F = " << arr.number_of_faces() << std::endl;
  return 0;
}
```

File: examples/Arrangement_on_surface_2/global_insertion.cpp

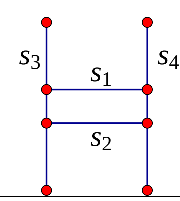
The number type used in the example above, *Quotient<MP_Float>*, is comprised of a numerator and a denominator of type *MP_Float*, namely floating-point numbers with unbounded mantissa. This number type is therefore capable of exactly computing the intersection points as long as the segment endpoints are given as floating-point numbers.

32.4.3 Removing Vertices and Edges

The free functions *remove_vertex()* and *remove_edge()* handle the removal of vertices and edges from an arrangement. The difference between these functions and the member functions of the *Arrangement_2* template having the same name is that they allow the merger of two curves associated with adjacent edges to form a single edge. Thus, they require that the traits class that instantiates the arrangement instance is a model of the refined *ArrangementXMonotoneTraits_2* concept (see Section 32.6).

The function *remove_vertex(arr, v)* removes the vertex v from the given arrangement *arr*, where v is either an isolated vertex or is a *redundant* vertex — namely, it has exactly two incident edges that are associated with two curves that can be merged to form a single x-monotone curve. If neither of the two cases apply, the function returns an indication that it has failed to remove the vertex.

The function *remove_edge(arr, e)* removes the edge e from the arrangement by simply calling *arr.remove_edge(e)* (see Section 32.2.3). In addition, if either of the end vertices of e becomes isolated or redundant after the removal of the edge, it is removed as well.



The following example demonstrates the usage of the free removal functions. In creates an arrangement of four line segment forming an H-shape with a double horizontal line. Then it removes the two horizontal edges and clears all redundant vertices, such that the final arrangement consists of just two edges associated with the vertical line segments:

```
#include <CGAL/Cartesian.h>
#include <CGAL/Arr_segment_traits_2.h>
#include <CGAL/Arrangement_2.h>
#include <CGAL/Arr_naive_point_location.h>

#include "arr_print.h"
```

```
typedef int                                              Number_type;
typedef CGAL::Cartesian<Number_type>                     Kernel;
typedef CGAL::Arr_segment_traits_2<Kernel>               Traits_2;
typedef Traits_2::Point_2                                Point_2;
typedef Traits_2::X_monotone_curve_2                     Segment_2;
typedef CGAL::Arrangement_2<Traits_2>                    Arrangement_2;
typedef Arrangement_2::Vertex_handle                     Vertex_handle;
typedef Arrangement_2::Halfedge_handle                   Halfedge_handle;
typedef CGAL::Arr_naive_point_location<Arrangement_2> Naive_pl;

int main ()
{
  // Create an arrangement of four line segments forming an H-shape:
  Arrangement_2   arr;
  Naive_pl        pl (arr);

  Segment_2       s1 (Point_2 (1, 3), Point_2 (4, 3));
  Halfedge_handle e1 = arr.insert_in_face_interior (s1,
   arr.unbounded_face());
  Segment_2       s2 (Point_2 (1, 4), Point_2 (4, 4));
  Halfedge_handle e2 = arr.insert_in_face_interior (s2,
   arr.unbounded_face());
  Segment_2       s3 (Point_2 (1, 1), Point_2 (1, 6));
  Segment_2       s4 (Point_2 (4, 1), Point_2 (4, 6));

  insert (arr, s3, pl);
  insert (arr, s4, pl);

  std::cout << "The initial arrangement:" << std::endl;
  print_arrangement (arr);

  // Remove the horizontal edge from the arrangement, and its end
   vertices:
  Vertex_handle   v1 = e1->source(), v2 = e1->target();
  arr.remove_edge (e1);
  remove_vertex (arr, v1);
  remove_vertex (arr, v2);

  // Remove the second horizontal edge e2 from the arrangement:
  remove_edge (arr, e2);

  std::cout << "The final arrangement:" << std::endl;
  print_arrangement (arr);
  return 0;
}
```

File: examples/Arrangement_on_surface_2/global_removal.cpp

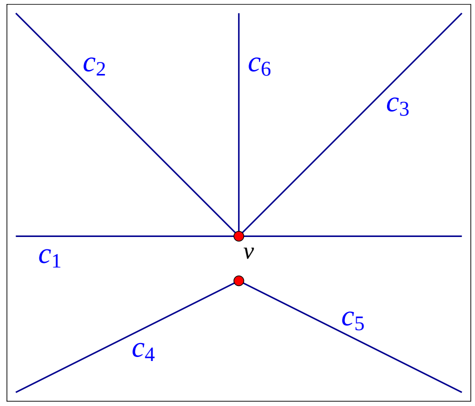


Figure 32.10: An arrangement of unbounded linear objects, as constructed in *unbounded_non_intersecting.cpp*.

32.5 Arrangements of Unbounded Curves

Previous sections dealt only with arrangements of line segments, namely of bounded curves. Such arrangements always have one unbounded face that contains all other arrangement features. This section explains how to construct arrangements of unbounded curves, such as lines and rays.

32.5.1 Basic Manipulation and Traversal Methods

Consider the arrangement induced by the two lines $y = x$ and $y = -x$. These two lines intersect at the origin, such that the arrangement contains a single vertex $v = (0,0)$, with four infinite rays emanating from it. Each ray corresponds to an arrangement edge, and these edges subdivide the plane into four unbounded faces. Consider a halfedge pair that represents one of the edges. The source vertex of one of these halfedges is v and its target is at infinity, while the other has its source at infinity and v is its target.

If e is an object of the nested type *Arrangement_2::Halfedge*, then the predicates *e.source_at_infinity()* and *e.target_at_infinity()* indicate whether the halfedge represents a curve with an infinite end. In general there is no need to access the source (or the target) of a halfedge if it lies at infinity, since this vertex is not associated with any valid point. Similarly, calling *arr.number_of_vertices()* for an arrangement object *arr* counts only the vertices associated with finite points, and ignores vertices at infinity (and the range *[vertices_begin(), vertices_end())* contains only finite vertices). The method *arr.number_of_vertices_at_infinity()* counts the number of vertices at infinity.

As mentioned above, arrangements of unbounded curves usually have more than one unbounded face. The function *arr.number_of_unbounded_faces()* returns the number of unbounded arrangement faces (Thus, *arr.number_of_faces() - arr.number_of_unbounded_faces()* is the number of bounded faces). The functions *arr.unbounded_faces_begin()* and *arr.unbounded_faces_end()* return iterators of type *Arrangement_2::Unbounded_face_iterator* that specify the range of unbounded faces. Naturally, the value-type of this iterator is *Arrangement_2::Face*.

The specialized insertion functions listed in Section 32.2.3 can also be used for inserting x-monotone unbounded curves, provided that they are interior-disjoint from any subcurve that already exists in the arrangement. For example, if you wish to insert a ray r emanating from $(0,0)$ in the direction of $(1,0)$, to the arrangement of $y = -x$ and $y = x$, you can use the function *arr.insert_from_left_vertex()*, as the left endpoint of r is already associated with an arrangement vertex. Other edge-manipulation functions can also be applied on edges associated with unbounded curves.

The following example demonstrates the use of the insertion function for pairwise interior-disjoint unbounded curves. In this example we use the traits class *Arr_linear_traits_2<Kernel>* to instantiate the *Arrangement_2* template. This traits class is capable of representing line segments as well as unbounded linear curves (namely lines and rays). Observe that objects of the type *X_monotone_curve_2* defined by this traits class are constructible from *Line_2*, *Ray_2*, and *Segment_2* objects, as defined in the instantiated kernel.

The first three curves are inserted using the special insertion functions for x-monotone curves whose location in the arrangement is known. Notice that inserting an unbounded curve in the interior of an unbounded face, or from an existing vertex that represents the bounded end of the curve, may cause an unbounded face to split (this is never the case when inserting a bounded curve — compare with Section 32.2.3). Then, three additional rays are inserted incrementally, using the insertion function for x-monotone curves whose interior is disjoint from all arrangement features. Finally, the program prints the size of the arrangement (compare to the illustration in Figure 32.10) and the outer boundaries of its six unbounded faces:

```
#include <CGAL/Simple_cartesian.h>
#include <CGAL/Arr_linear_traits_2.h>
#include <CGAL/Arrangement_2.h>

typedef int                                           Number_type;
typedef CGAL::Simple_cartesian<Number_type>           Kernel;
typedef CGAL::Arr_linear_traits_2<Kernel>             Traits_2;
typedef Traits_2::Point_2                             Point_2;
typedef Traits_2::Segment_2                           Segment_2;
typedef Traits_2::Ray_2                               Ray_2;
typedef Traits_2::Line_2                              Line_2;
typedef Traits_2::X_monotone_curve_2                  X_monotone_curve_2;
typedef CGAL::Arrangement_2<Traits_2>                 Arrangement_2;
typedef Arrangement_2::Vertex_handle                  Vertex_handle;
typedef Arrangement_2::Halfedge_handle                Halfedge_handle;

int main ()
{
  Arrangement_2      arr;

  // Insert a line in the (currently single) unbounded face of the
   arrangement,
  // then split it into two at (0,0). Assign v to be the split point.
  X_monotone_curve_2 c1 = Line_2 (Point_2 (-1, 0), Point_2 (1, 0));
  Halfedge_handle    e1 = arr.insert_in_face_interior (c1,

   arr.unbounded_face());

  X_monotone_curve_2 c1_left = Ray_2 (Point_2 (0, 0), Point_2 (-1, 0));
  X_monotone_curve_2 c1_right = Ray_2 (Point_2 (0, 0), Point_2 (1, 0));

  e1 = arr.split_edge (e1, c1_left, c1_right);
  Vertex_handle      v = e1->target();

  CGAL_assertion (! v->is_at_open_boundary());

  // Add two more rays using the specialized insertion functions.
  X_monotone_curve_2 c2 = Ray_2 (Point_2 (0, 0), Point_2 (-1, 1));
  X_monotone_curve_2 c3 = Ray_2 (Point_2 (0, 0), Point_2 (1, 1));

  arr.insert_from_right_vertex (c2, v);
```

```
  arr.insert_from_left_vertex (c3, v);

  // Insert three more interior-disjoint rays.
  X_monotone_curve_2 c4 = Ray_2 (Point_2 (0, -1), Point_2 (-2, -2));
  X_monotone_curve_2 c5 = Ray_2 (Point_2 (0, -1), Point_2 (2, -2));
  X_monotone_curve_2 c6 = Ray_2 (Point_2 (0, 0), Point_2 (0, 1));

  insert_non_intersecting_curve (arr, c4);
  insert_non_intersecting_curve (arr, c5);
  insert_non_intersecting_curve (arr, c6);

  // Print out the size of the resulting arrangement.
  std::cout << "The arrangement size:" << std::endl
            << "   V = " << arr.number_of_vertices()
            << " (plus " << arr.number_of_vertices_at_infinity()
            << " at infinity)"
            << ",  E = " << arr.number_of_edges()
            << ",  F = " << arr.number_of_faces()
            << " (" << arr.number_of_unbounded_faces() << " unbounded)"
            << std::endl << std::endl;

  // Print the outer CCBs of the unbounded faces.
  Arrangement_2::Face_const_iterator            fit;
  Arrangement_2::Ccb_halfedge_const_circulator  first, curr;
  Arrangement_2::Halfedge_const_handle          he;
  int                                           k = 1;

  for (fit = arr.faces_begin(); fit != arr.faces_end(); ++fit, k++) {
    if (! fit->is_unbounded())
      continue;

    std::cout << "Face no. " << k << ": ";
    curr = first = fit->outer_ccb();
    if (! curr->source()->is_at_open_boundary())
      std::cout << "(" << curr->source()->point() << ")";
    do {
      he = curr;
      if (! he->is_fictitious())
        std::cout << "   [" << he->curve() << "]   ";
      else
        std::cout << "   [ ... ]   ";

      if (! he->target()->is_at_open_boundary())
        std::cout << "(" << he->target()->point() << ")";

      ++curr;
    } while (curr != first);
    std::cout << std::endl;
  }

  return 0;
}
```

File: examples/Arrangement_on_surface_2/unbounded_non_intersecting.cpp

32.5.2 Free Functions

In principle, all queries and operations that relate to arrangements of bounded curves can also be applied to arrangements of unbounded curves. For example, it is possible to issue point-location and vertical ray-shooting queries (see also Section 32.3) on arrangements of lines, where the only restriction is that the query point has finite coordinates.[11]

In the following example we show how an arrangement of unbounded lines is utilized to solve the following problem: Given a set of points, does the set contain at least three collinear points? In this example a set of input points is read from a file. The file *points.dat* is used by default. It contains definitions of 100 points randomly selected on the grid $[-10000, 10000] \times [-10000, 10000]$. We construct an arrangement of the dual lines, where the line p^* dual to the point $p = (p_x, p_y)$ is given by the equation $y = p_x * x - p_y$, and check whether three (or more) of the dual lines intersect at a common point, by searching for a (dual) vertex, whose degree is greater than 4. If such a vertex exists, then there are at least three dual lines that intersect at a common point, which implies that there are at least three collinear points.

```
#include "arr_rational_nt.h"
#include <CGAL/Cartesian.h>
#include <CGAL/Arr_linear_traits_2.h>
#include <CGAL/Arrangement_2.h>
#include <cstdlib>

typedef CGAL::Cartesian<Number_type>                    Kernel;
typedef CGAL::Arr_linear_traits_2<Kernel>               Traits_2;
typedef Traits_2::Point_2                               Point_2;
typedef Traits_2::Line_2                                Line_2;
typedef Traits_2::X_monotone_curve_2                    X_monotone_curve_2;
typedef CGAL::Arrangement_2<Traits_2>                   Arrangement_2;

int main (int argc, char *argv[])
{
  // Get the name of the input file from the command line, or use the
   default
  // points.dat file if no command-line parameters are given.
  const char * filename = (argc > 1) ? argv[1] : "points.dat";

  // Open the input file.
  std::ifstream     in_file (filename);

  if (! in_file.is_open()) {
    std::cerr << "Failed to open " << filename << "!" << std::endl;
    return 1;
  }

  // Read the points from the file, and consturct their dual lines.
  // The input file format should be (all coordinate values are
   integers):
  // <n>                                  // number of point.
```

[11]Currently, all point-location strategies except the trapezoidal RIC point-location strategy are capable of handling arrangements of unbounded curves.

```
// <x_1> <y_1>                              // point #1.
// <x_2> <y_2>                              // point #2.
//   :       :         :        :
// <x_n> <y_n>                              // point #n.
std::vector<Point_2>            points;
std::list<X_monotone_curve_2>  dual_lines;

unsigned int n;
in_file >> n;
points.resize(n);
unsigned int k;
for (k = 0; k < n; ++k) {
  int px, py;
  in_file >> px >> py;
  points[k] = Point_2 (px, py);

  // The line dual to the point (p_x, p_y) is y = p_x*x - p_y,
  // or: p_x*x - y - p_y = 0:
  dual_lines.push_back (Line_2 (Number_type(px),
                                Number_type(-1),
                                Number_type(-py)));
}
in_file.close();

// Construct the dual arrangement by aggragately inserting the lines.
Arrangement_2      arr;

insert (arr, dual_lines.begin(), dual_lines.end());

std::cout << "The dual arrangement size:" << std::endl
          << "V = " << arr.number_of_vertices()
          << " (+ " << arr.number_of_vertices_at_infinity()
          << " at infinity)"
          << ",  E = " << arr.number_of_edges()
          << ",  F = " << arr.number_of_faces()
          << " (" << arr.number_of_unbounded_faces()
          << " unbounded)" << std::endl;

// Look for a vertex whose degree is greater than 4.
Arrangement_2::Vertex_const_iterator   vit;
bool                                   found_collinear = false;

for (vit = arr.vertices_begin(); vit != arr.vertices_end(); ++vit) {
  if (vit->degree() > 4) {
    found_collinear = true;
    break;
  }
}
if (found_collinear)
  std::cout << "Found at least three collinear points in the input
 set."
            << std::endl;
else
  std::cout << "No three collinear points are found in the input set."
```

```
                << std::endl;

  // Pick two points from the input set, compute their midpoint and
   insert
  // its dual line into the arrangement.
  Kernel                 ker;
  const int              k1 = std::rand() % n, k2 = (k1 + 1) % n;
  Point_2                p_mid = ker.construct_midpoint_2_object()
   (points[k1],

   points[k2]);
  X_monotone_curve_2 dual_p_mid = Line_2 (Number_type(p_mid.x()),
                                          Number_type(-1),
                                          Number_type(-p_mid.y()));

  insert (arr, dual_p_mid);

  // Make sure that we now have three collinear points.
  found_collinear = false;
  for (vit = arr.vertices_begin(); vit != arr.vertices_end(); ++vit) {
    if (vit->degree() > 4) {
      found_collinear = true;
      break;
    }
  }
  CGAL_assertion (found_collinear);
  return (0);
}
```

File: examples/Arrangement_on_surface_2/dual_lines.cpp

Note that there are no three collinear points among the points defined in the input file *points.dat*. In the second part of the example the existence of collinearity is forced and verified as follows. A line dual to the midpoint of two randomly selected points is introduced, and inserted into the arrangement. This operation is followed by a test that verifies that a vertex of degree greater than 4 exists. This implied that collinearity indeed exists as explained above.

advanced

32.5.3 Representation of Unbounded Arrangements

Given a set $\mathcal{C}$ of unbounded curves, a simple approach for representing the arrangement induced by $\mathcal{C}$ would be to clip the unbounded curves using an axis-parallel rectangle that contains all finite curve endpoints and intersection points between curves in $\mathcal{C}$. This process would result in a set $\mathcal{C}$ of bounded curves (line segments if $\mathcal{C}$ contains lines and rays), and it would be straightforward to compute the arrangement induced by this set. However, we would like to operate directly on the unbounded curves without having to preprocess them. Therefore, we use an implicit bounding rectangle embedded in the DCEL structure. Figure 32.11 shows the arrangement of four lines that subdivide the plane into eight unbounded faces and two bounded ones. Notice that in this case the unbounded faces have outer boundaries, and the halfedges along these outer CCBs are drawn as arrows. The bounding rectangle is drawn with a dashed line. The vertices $v_1, v_2, \ldots, v_8$, which represent the unbounded ends of the four lines, and lie on the bounding rectangle, actually exist at infinity, and the halfedges

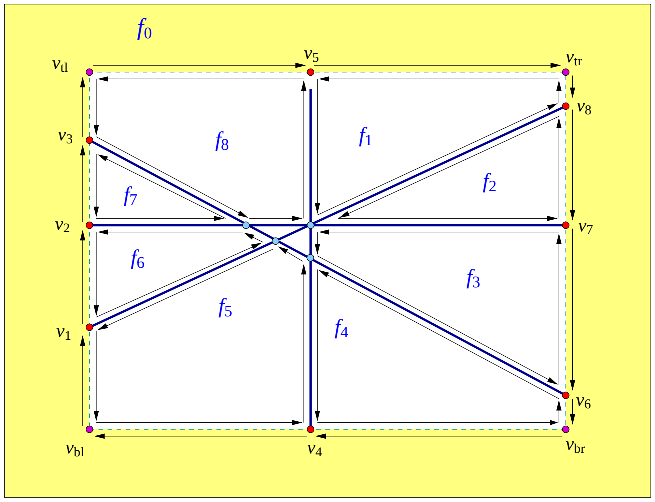


Figure 32.11: A DCEL representing an arrangement of four lines. Halfedges are drawn as thin arrows. The vertices $v_1, \ldots, v_8$ lie at infinity, and are not associated with valid points. The halfedges that connect them are fictitious, and are not associated with concrete curves. The face denoted f_0 (lightly shaded) is the fictitious "unbounded face" which lies outside the bounding rectangle (dashed) that bounds the actual arrangement. The four fictitious vertices v_{bl}, v_{tl}, v_{br} and v_{tr} represent the four corners of the bounding rectangle.

connecting them are *fictitious*, and represent portions of the bounding rectangle. Note that the outer CCBs of the unbounded faces contain fictitious halfedges. The twins of these halfedges form together one connected component that corresponds to the entire bounding rectangle, which forms a single hole in a face f_0. We say that f_0 is *fictitious*, as it does not correspond to a real two-dimensional cell of the arrangement.

Observe that there are four extra vertices at infinity that do not lie on any curve; they are denoted as v_{bl}, v_{tl}, v_{br}, and v_{tr}, and represent the bottom-left, top-left, bottom-right, and top-right corners of the bounding rectangle, respectively. Similarly, there are fictitious halfedges that lie on the top, the bottom, the left, or the right edge of the bounding rectangle. When the arrangement is empty, there are exactly four pairs of fictitious halfedges, that divide the plane into two faces, namely a fictitious face lying outside of the bounding rectangle and a single unbounded face bounded by the bounding rectangle.

Summarizing the above, there are four types of arrangement vertices, which differ from one another by their location with respect to the bounding bounding rectangle:

1. A vertex, associated with a point in $\mathbb{R}^2$ whose coordinates are bounded. Such a vertex always lies inside the bounding rectangle.
2. A vertex that represents an unbounded end of an x-monotone curve that is defined at $x = -\infty$ or at $x = \infty$. In case of a horizontal line or a curve with a horizontal asymptote, the y-coordinate of the curve end may be finite (see for example the vertices v_2 and v_7 in Figure 32.11), but in general the curve end also goes to $y = \pm\infty$ (see for instance the vertices v_1, v_3, v_6 and v_8 in Figure 32.11). For our convenience, we will always take a "tall" enough bounding rectangle and treat such vertices as lying on either the left or right rectangle edges (that is, if a curve is defined at $x = -\infty$, its left end will be represented by a vertex on the left edge of the bounding rectangle, and if it is defined at $x = \infty$, its right end will be represented by a vertex of the right edge).
3. A vertex that represent the unbounded end of a vertical line or of a curve with a vertical asymptote (finite x-coordinate and an unbounded y-coordinate). Such a vertex always lies on one of the horizontal edges of the bounding rectangle (either the bottom one if $y = -\infty$, or the top one if $y = \infty$). The vertices v_4 and v_5 in Figure 32.11 are of this type.

4. The fictitious vertices that represent the four corners of the bounding bounding rectangle.

A vertex (at infinity) of Type 2 or Type 3 above always has three incident edges: one concrete edge that is associated with an unbounded portion of an x-monotone curve, and two fictitious edges connecting the vertex to its neighboring vertices at infinity. Fictitious vertices (of type 4 above) have exactly two incident edges. See Section 32.6 on how the traits-class interface helps imposing the fact that we never have more than one curve incident to any true vertex at infinity.

The nested types defined in the *Arrangement_2* class support the following methods, in addition to the ones listed in Section 32.2.2:

- The *Vertex* class provides three-valued predicates *parameter_space_in_x()* and *parameter_space_in_y()*, which return the location of the geometric embedding of the vertex in the parameter space. In particular, the former returns *ARR_LEFT_BOUNDARY*, *ARR_INTERIOR*, or *ARR_RIGHT_BOUNDARY*, and the latter returns *ARR_BOTTOM_BOUNDARY*, *ARR_INTERIOR*, or *ARR_TOP_BOUNDARY*. As the package currently supports only the case where the parameter space is the compactified plane, the former returns *ARR_INTERIOR* if the x-coordinate associated with the vertex is finite, *ARR_LEFT_BOUNDARY* if it is $-\infty$, and *ARR_RIGHT_BOUNDARY* if it is ∞. The latter returns *ARR_INTERIOR* if the y-coordinate associated with the vertex is finite, *ARR_BOTTOM_BOUNDARY* if it is $-\infty$, and *ARR_TOP_BOUNDARY* if it is ∞. The Boolean predicate *is_at_open_boundary()* is also provided. You can access the point associated with a vertex only if it is not a vertex at an open boundary (recall that a vertex at an open boundary is not associated with a *Point_2* object).
- The nested *Halfedge* class provides the Boolean predicate *is_fictitious()*. The x-monotone curve associated with a halfedge can be accessed by the *curve()* method only if the halfedge is not fictitious.
- The nested *Face* class provides the Boolean predicate *f.is_fictitious()*. The method *outer_ccb()* has the precondition that the face is not fictitious. Note that non-fictitious unbounded faces always have valid CCBs (although this CCB may comprise only fictitious halfedge in case the arrangement contains only bounded curves).

The method *arr.number_of_edges()* does not count the number of fictitious edges, (which is always *arr.number_of_vertices_at_infinity()* + *4*), and the iterators returned by *arr.edges_begin()* and *arr.edges_end()* specify a range of non-fictitious edges. Similarly, *arr.number_of_faces()* does not count the fictitious face. However, the *Ccb_halfedge_circulator* of the outer boundary of an unbounded face or the *Halfegde_around_vertex_circulator* of a vertex at infinity do traverse fictitious halfedges. For example, it is possible to traverse the outer boundaries of the unbounded arrangement edges using the following procedure:

```
Arrangement_2::Unbounded_face_const_iterator  fit;
Arrangement_2::Ccb_halfedge_const_circulator  first, curr;
Arrangement_2::Halfedge_const_handle          he;
int                                           k = 1;

for (fit = arr.unbounded_faces_begin();
     fit != arr.unbounded_faces_end(); ++fit, k++)
{
  std::cout << "Unbounded face no. " << k << ": ";
  curr = first = fit->outer_ccb();
  if (! curr->source()->is_at_infinity())
    std::cout << "(" << curr->source()->point() << ")";
  do
  {
    he = curr;
```

```
      if (! he->is_fictitious())
        std::cout << "   [" << he->curve() << "]   ";
      else
        std::cout << "   [ ... ]   ";

      if (! he->target()->is_at_infinity())
        std::cout << "(" << he->target()->point() << ")";

      ++curr;
    } while (curr != first);
    std::cout << std::endl;
  }
```

advanced

32.6 Traits Classes

As mentioned in the introduction of this chapter, the traits class encapsulates the definitions of the geometric entities and implements the geometric predicates and constructions needed by the *Arrangement_2* class and by its peripheral algorithms. We also mention throughout the chapter that there are different levels of requirements from the traits class, namely the traits class can model different concept refinement-levels.

32.6.1 The Hierarchy of Traits-Class Concepts

The Basic Concept

A model of the basic concept, *ArrangementBasicTraits_2*, needs to define the types *Point_2* and *X_monotone_curve_2*, where objects of the first type are the geometric mapping of arrangement vertices, and objects of the latter type are the geometric mapping of edges. Such a model has to support in addition the following set of operations:

Compare_x_2: Compares the x-coordinates of two points.

Compare_xy_2: Compares two points lexicographically, by their x-coordinates and then (in case of equality) by their y-coordinates.

Construct_min_vertex_2,*Construct_max_vertex_2*: Returns the left endpoint (similarly, the right endpoint) of an x-monotone curve.

Compare_y_at_x_2: Given an x-monotone curve c and a point p that lies in its x-range, this predicate determines whether p lies below, above or on c.

Compare_y_at_x_right_2: Given two x-monotone curves c_1 and c_2 that share a common left endpoint p, this predicate determines whether c_1 lies above or under c_2 immediately to the right of p, or whether the two curves coincide there.

Equal_2: Checks two points and two curves for equality (two curves are equal if their graph is the same).

Is_vertical_2: Determines whether an x-monotone curve is vertical.

Each model of the concept *ArrangementBasicTraits_2* needs to define a tag named *Has_left_category*. It determines whether the traits class supports the following predicate:

Compare_y_at_x_left_2: Given two x-monotone curves c_1 and c_2 that share a common right endpoint p, this predicate determines whether c_1 lies above or under c_2 immediately to the left of p, or whether the two curves coincide there.

This predicate is optional, as it can be answered using the other traits-class primitives, and we wish to alleviate the need to implement an extra method that is not absolutely necessary. However, as implementing the predicate directly may prove to be more efficient, the traits-class implementer may choose to provide it.

The basic set of predicates is sufficient for constructing arrangements of x-monotone curves that do not reach or approach the boundary of the parameter space. The nature of the input curves, i.e., whether some of them are expected to reach or approach the left, right, bottom, or top side of the boundary of the parameter space, must be conveyed by the traits class. This is done through the definition of four additional nested types, namely *Left_side_category*, *Right_side_category*, *Bottom_side_category*, and *Top_side_category*. Each of those types must be convertible to the type *Arr_oblivious_side_tag* for the class to be a model of the concept *ArrangementBasicTraits_2*.

The Landmarks Concept

The landmark point-location strategy (see Section 32.3.1) needs its associated arrangement to be instantiated with a model of the refined *ArrangementLandmarkTraits_2* traits concept. A model of this concept must define a fixed precision number type (typically *double*) and support the additional operations:

Approximate_2: Given a point p, approximate the x and y-coordinates of p using the fixed precision number type. We use this operation for approximate computations—there are certain operations in the search for the location of the point that need not be exact and we can perform them faster than other operations.

Construct_x_monotone_curve_2: Given two points p_1 and p_2, this predicate constructs an x-monotone curve connecting p_1 and p_2.

Supporting Intersecting x-Monotone Curves

A traits class that models the *ArrangementXMonotoneTraits_2* concept, which refines the *ArrangementBasicTraits_2* concept, has to support the following functions:

Intersection_2: Computes all intersection points and overlapping sections of two given x-monotone curves. If possible, computes also the multiplicity of each intersection point.[12] Knowing the multiplicity of an intersection point is not required, but it can speed up the arrangement construction.

Split_2: Splits an x-monotone curve c into two subcurves at a point p lying in the interior of c.

Are_mergeable_2: Given two x-monotone curve c_1 and c_2 that share a common endpoint, this predicate determines whether c_1 and c_2 are *mergeable*, that is, whether they can be merged to form a single continuous x-monotone curve of the type supported by the traits class.

Merge_2: Merges two mergeable x-monotone curves.

Using a model of the *ArrangementXMonotoneTraits_2*, it is possible to construct arrangements of sets of x-monotone curves (and points) that may intersect one another.

[12] If the two curves intersect at a point p but have different tangents, p is of multiplicity 1. If the tangents are also equal but the their curvatures are not the same, p is of multiplicity 2, etc.

Supporting Arbitrary Curves

The concept *ArrangementTraits_2* refines the *ArrangementXMonotoneTraits_2* concept by adding the notion of a general, not necessarily x-monotone (and not necessarily continuous) curve. A model of this concept must define the *Curve_2* type and support the subdivision of a curve into a set of continuous x-monotone curves and isolated points using the predicate *Make_x_monotone_2*. For example, the curve $C: \ (x^2+y^2)(x^2+y^2-1)=0$ is the unit circle (the loci of all points for which $x^2+y^2=1$) with the origin $(0,0)$ as a singular point in its interior. C should therefore be divided into two circular arcs (the upper part and the lower part of the unit circle) and a single isolated point.

Note that the refined model *ArrangementTraits_2* is required only when using the free *insert()* functions (see Section 32.4), which accept a *Curve_2* object in the incremental version, or a range of *Curve_2* objects in the aggregated version. In all other cases it is sufficient to use a model of the *ArrangementXMonotoneTraits_2* concept.

Supporting Unbounded Curves

An arrangement that supports unbounded x-monotone curves maintains an implicit bounding rectangle in the DCEL structure; see Section 32.5.3. The unbounded ends of vertical rays, vertical lines, and curves with vertical asymptotes are represented by vertices that lie on the bottom or top sides of this bounding rectangle. These vertices are not associated with points, but are associated with (finite) x-coordinates. The unbounded ends of all other curves are represented by vertices that lie on the left or right sides of this bounding rectangle. These vertices are not associated with points either. Edges connect these vertices and the four vertices that represents the corners of this bounding rectangle to form the rectangle.

Several predicates are required to handle x-monotone curves that approach infinity and thus approach the boundary of the parameter space. These predicates are sufficient to handle not only curves embedded in an unbounded parameter space, but also curves embedded in a bounded parameter space with open boundaries. Let b_l and b_r denote the x-coordinates of the left and right boundaries of the parameter space, respectively. Let b_b and b_t denote the y-coordinates of the bottom and top boundaries of the parameter space, respectively. Recall that currently the general code of the arrangement only supports the case where the parameter space is the entire compactified plane, thus $b_l = b_b = -\infty$ and $b_r = b_t = +\infty$. Nonetheless, when the parameter space is bounded, it is the exact geometric embedding of the implicit bounding rectangle. In the following we assume that an x monotone curve C can be considered as a parametric curve $C(t) = (X(t), Y(t))$ defined over a closed, open, or half open interval with endpoints 0 and 1.

Models of the concept *ArrangementOpenBoundaryTraits_2* handle curves that approach the boundary of the parameter space. This concept refines the concept *ArrangementBasicTraits_2*. The arrangement template instantiated with a traits class that models this concept can handle curves that are unbounded in any direction. If some curves inserted into an arrangement object are expected to be unbounded, namely, there exists $d \in \{0,1\}$ such that $\lim_{t\rightarrow d} X(t) = \pm\infty$ or $\lim_{t\rightarrow d} y(t) = \pm\infty$ holds for at least one input curve $C(t) = (X(t), Y(t))$, the arrangement template must be instantiated with a model of the *ArrangementOpenBoundaryTraits* concept.[13]

All the four types *Left_side_category*, *Right_side_category*, *Bottom_side_category*, and *Top_side_category* nested in a model of the concept *ArrangementOpenBoundaryTraits* must be convertible to *Arr_open_side_tag*.[14] For example, the *Arr_rational_function_traits_2* traits-model supports unbounded curves; see Section 32.6.6. Thus, all four nested types are defined as *Arr_open_side_tag*. Adversely, all four types nested in the *Arr_segment_traits_2* traits-model (see Section 32.6.2) are defined as *Arr_oblivious_side_tag*, as segments are always

[13] We intend to enhance the arrangement template to handle curves confined to a bounded yet open parameter space. A curve that reaches the boundary of the parameter space in this case is bounded and open.

[14] The tags *Arr_oblivious_side_tag* and *Arr_open_side_tag* are only two out of a larger number of options for the side categories included in major extension the code is going through.

bounded.[15]

A model of the concept *ArrangementOpenBoundaryTraits_2* must provide the additional predicates listed below. x-coordinates and y-coordinates are differently handled. This asymmetry is brought on by the various algorithms applied to arrangements, the input and output arguments of which are x-monotone curves. Indeed, all curves maintained by any arrangement are continuous weakly x-monotone curves. A non x-monotone curve is divided into x-monotone sub curves (and perhaps points) before it is inserted into an arrangement. This asymmetry is also reflected in the additional predicates listed below.

Parameter_space_in_x_2: Given a parametric x-monotone curve $C(t) = (X(t),Y(t))$ and an enumerator that specifies either the minimum end or the maximum end of the curve, and thus maps to a parameter value $d \in \{0,1\}$, this predicate determines the location of the curve end along the x-dimension. Formally, the predicate determines whether $\lim_{t\to d} X(t)$ evaluates to b_l, b_r, or a value in between.

Compare_y_near_boundary_2: Given two x-monotone curves C_1 and C_2 and an enumerator i that specifies either the minimum ends or the maximum ends of the two curves, this predicate compares the y-coordinates of the curves near their respective ends. That is, the predicate compares the y-coordinates of the vertical projection of a point p onto C_1 and onto C_2. If the enumerator i specifies the minimum ends, the curves must approach the left boundary-side. In this case p is located far to the left, such that the result is invariant under a translation of p farther to the left. If i specifies the maximum ends, the curves must approach the right boundary-side. In that case p is located far to the right in a similar manner.

Parameter_space_in_y_2: Given a parametric x-monotone curve $C(t) = (X(t),Y(t))$ and an enumerator that specifies either the minimum end or the maximum end of the curve, and thus maps to a parameter value $d \in \{0,1\}$, this predicate determines the location of the curve end along the y-dimension. Formally, the predicate determines whether $\lim_{t\to d} Y(t)$ evaluates to b_b, b_t, or a value in between.

Compare_x_at_limit_2: Two versions of this predicate are provided: (i) Given a point p, a parametric x-monotone curve $C(t) = (X(t),Y(t))$, and an enumerator that specifies either the minimum end or the maximum end of the curve, and thus maps to a parameter value $d \in \{0,1\}$, this predicate compares the x-coordinate of p and $\lim_{t\to d} X(t)$. If the parameter space is unbounded, a precondition assures that C has a vertical asymptote at its d-end; that is $\lim_{t\to d} X(t)$ is finite. (ii) Given two parametric x-monotone curves $C_1(t) = (X_1(t),Y_1(t))$ and $C_2(t) = (X_2(t),Y_2(t))$ and two enumerators that specify either the minimum end or the maximum end of each curve, and thus map to parameter values $d_1 \in \{0,1\}$ and $d_2 \in \{0,1\}$ for C_1 and for C_2, respectively, this predicate compares $\lim_{t\to d_1} X_1(t)$ and $\lim_{t\to d_2} X_2(t)$. If the parameter space is unbounded, a precondition assures that C_1 and C_2 have vertical asymptote at their respective ends; that is $\lim_{t\to d_1} X_1(t)$ and $\lim_{t\to d_2} X_2(t)$ are finite.

Compare_x_near_limit_2: Given two x-monotone curves C_1 and C_2 and an enumerator i that specifies either the minimum ends or the maximum ends of the two curves, this predicate compares the x-coordinates of the curves near their respective ends. That is, the predicate compares the x-coordinates of the horizontal projection of a point p onto C_1 and onto C_2. If the parameter space is unbounded, a precondition assures that C_1 and C_2 have vertical asymptote at their respective ends. Furthermore, both curves approach the same boundary-side, either the bottom or the top, at their respective ends. If both curves approach the bottom boundary-side, p is located far to the bottom, such that the result is invariant under a translation of p farther to the bottom. If both curves approach the top boundary-side, p is located far to the top in a similar manner. Another precondition assures that the x-coordinates of the limits of the curves at their respective ends are equal. That is, the predicate *Compare_x_at_limit_2* applied to C_1, C_2, and i evaluates to *EQUAL*.

In the rest of this section we review the traits classes included in the public distribution of CGAL, that handle line segments, polylines, conic arcs, rational functions, and arcs of Bézier and algebraic curves. The last subsection overviews decorators for geometric traits classes distributed with CGAL, which extend other geometric traits-class by attaching auxiliary data with the geometric objects.

[15] We intend to introduce more concepts that require only a subset of the categories to be convertible to *Arr_open_side_tag*.

32.6.2 Traits Classes for Line Segments and Linear Objects

The *Arr_segment_traits_2<Kernel>* class used so far in most example programs in this chapter is a model of the concepts *ArrangementTraits_2*, *ArrangementLandmarkTraits_2*, and *ArrangementDirectionalXMonotoneTraits_2*; the later enables Boolean set operations. It is parameterized by a geometric kernel and uses the *Kernel::Point_2* type as it point type. However, neither the *Curve_2* nor the *X_monotone_curve_2* types are identical to the *Kernel::Segment_2* type. A kernel segment is typically represented by its two endpoints, and these may have a large bit-size representation, if the segment is intersected and split several times (in comparison with the representation of its original endpoints). The large representation may significantly slow down the various traits-class operations involving such a segment. In contrast, the *Arr_segment_traits_2* represents a segment using its supporting line and the two endpoints, such that most computations are performed on the supporting line, which never changes as the segment is split. It also caches some additional information with the segment to speed up various predicates. An *X_monotone_curve_2* object can still be constructed from two endpoints or from a kernel segment. Moreover, an *X_monotone_curve_2* instance can also be casted or assigned to a *Kernel::Segment_2* object. The two types are thus fully convertible to one another.

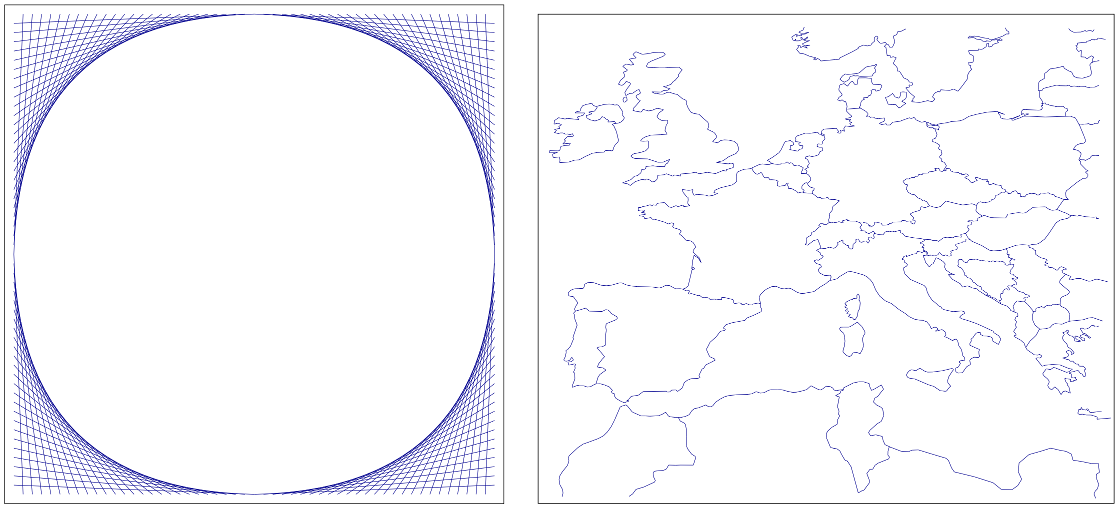
The *Arr_segment_traits_2<Kernel>* class is very efficient for maintaining arrangements of a large number of intersecting line segments, especially if it is instantiated with the appropriate geometric kernel. Using *Cartesian< Gmpq>* as the kernel type is generally a good choice; the coordinates of the segment endpoints are represented as exact rational numbers, and this ensures the robustness and correctness of any computation. However, if the GMP library is not installed, it is possible to use the *Quotient<MP_Float>* number-type, provided by the support library of CGAL, which is somewhat less efficient.[16]

Exact computations are of course less efficient, compared to machine-precision floating-point arithmetic, so constructing an arrangement using the *Cartesian<Gmpq>* kernel (or, similarly, *Cartesian<Quotient<MP_Float> >*) is several times slower in comparison to a *Simple_cartesian<double>* kernel. However, in the latter case the correctness of the computation results is not guaranteed. In many cases it is possible to use *filtered* computations and benefit from both approaches, namely achieve fast running times with guaranteed results. In case we handle a set of line segments that have no degeneracies, namely no two segments share a common endpoint, and no three segments intersect at a common point — or alternatively, degeneracies exist but their number is relatively small — then filtered computation incur only a minor overhead compared to the floating-point arithmetic, while ensuring the correctness of the computation results.

In the following example we use the predefined *Exact_predicates_exact_constructions_kernel* for instantiating our segment-traits class. This kernel use interval arithmetic to filter the exact computations. The program reads a set of line segments with integer coordinates from a file and computes their arrangement. By default it opens the *fan_grids.dat* input-file, located in the examples folder, which contains 104 line segments that form four "fan-like" grids and induce a dense arrangement, as illustrated in Figure 32.12(a):

```
#include <CGAL/Exact_predicates_exact_constructions_kernel.h>
#include <CGAL/Arr_segment_traits_2.h>
#include <CGAL/Arrangement_2.h>
#include <CGAL/Timer.h>
#include <list>
#include <fstream>

typedef CGAL::Exact_predicates_exact_constructions_kernel  Kernel;
typedef Kernel::FT                                         Number_type;
typedef CGAL::Arr_segment_traits_2<Kernel>                 Traits_2;
typedef Traits_2::Point_2                                  Point_2;
```

[16] Many of the example programs in the rest of the chapter include a header file named *arr_rational_nt.h*, which defines a type named *Number_type* as either *Gmpq* or *Quotient<MP_Float>*, depending on whether GMP is installed or not.



(a) (b)

Figure 32.12: (a) An arrangement of 104 line segments from the input file *fan_grids.dat*. (b) An arrangement of more than 3000 interior disjoint line segments, defined in the input file *Europe.dat*.

```
typedef Traits_2::X_monotone_curve_2                          Segment_2;
typedef CGAL::Arrangement_2<Traits_2>                         Arrangement_2;

int main (int argc, char *argv[])
{
  // Get the name of the input file from the command line, or use the
   default
  // fan_grids.dat file if no command-line parameters are given.
  const char * filename = (argc > 1) ? argv[1] : "fan_grids.dat";

  // Open the input file.
  std::ifstream     in_file (filename);

  if (! in_file.is_open()) {
    std::cerr << "Failed to open " << filename << " ..." << std::endl;
    return (1);
  }

  // Read the segments from the file.
  // The input file format should be (all coordinate values are
   integers):
  // <n>                              // number of segments.
  // <sx_1> <sy_1>  <tx_1> <ty_1>     // source and target of segment
   #1.
  // <sx_2> <sy_2>  <tx_2> <ty_2>     // source and target of segment
   #2.
  //   :      :        :      :
  // <sx_n> <sy_n>  <tx_n> <ty_n>     // source and target of segment
   #n.

  std::list<Segment_2>  segments;
```

```
  unsigned int n;
  in_file >> n;
  unsigned int i;
  for (i = 0; i < n; ++i) {
    int sx, sy, tx, ty;
    in_file >> sx >> sy >> tx >> ty;
    segments.push_back (Segment_2 (Point_2 (Number_type(sx),
   Number_type(sy)),
                                   Point_2 (Number_type(tx),
   Number_type(ty))));
  }
  in_file.close();

  // Construct the arrangement by aggregately inserting all segments.
  Arrangement_2                  arr;
  CGAL::Timer                    timer;

  std::cout << "Performing aggregated insertion of "
            << n << " segments." << std::endl;

  timer.start();
  insert (arr, segments.begin(), segments.end());
  timer.stop();

  // Print the arrangement dimensions.
  std::cout << "V = " << arr.number_of_vertices()
            << ",  E = " << arr.number_of_edges()
            << ",  F = " << arr.number_of_faces() << std::endl;

  std::cout << "Construction took " << timer.time()
            << " seconds." << std::endl;

  return 0;
}
```

File: examples/Arrangement_on_surface_2/predefined_kernel.cpp

The arrangement package also offers a simpler alternative segment-traits class. The traits class *Arr_non_caching_segment_basic_traits_2<Kernel>* models the *ArrangementBasicTraits_2* concept. It uses *Kernel::Point_2* as its point type and *Kernel::Segment_2* as its x-monotone curve type. As this traits class does not support intersecting and splitting segments, the kernel representation is sufficient. It is still less efficient than *Arr_segment_traits_2* for constructing arrangements of pairwise disjoint line segments in many cases, as it performs no caching at all, but using this traits class may be preferable as it reduces the memory consumption a bit, since no extra data is stored with the line segments.

The class *Arr_non_caching_segment_traits_2<Kernel>* inherits from *Arr_non_caching_segment_basic_traits_2< Kernel>* and extends it to be a model of the concepts *ArrangementTraits_2*, *ArrangementLandmarkTraits_2*,and *ArrangementDirectionalXMonotoneTraits_2*. It may thus be used to construct arrangement of intersecting line segments, but as explained above, for efficiency reasons it is recommended to use it only when the arrangement is very sparse and contains hardly any intersection points.

In the following example we read an input file containing a set of line segments that are pairwise disjoint in their interior. As the segments do not intersect, no new points are constructed and we can instantiate the *Arr_non_caching_segment_traits_basic_2<Kernel>* class-template with the predefined *Exact_predicates_inexact_constructions_kernel*. Note that we use the *insert_non_intersecting_curves()* function to construct the arrangement. By default, the example opens the *Europe.dat* input-file, located in the examples folder, which contains more than 3000 line segments with floating-point coordinates that form the map of Europe, as depicted in Figure 32.12(b):

```
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/Arr_non_caching_segment_basic_traits_2.h>
#include <CGAL/Arrangement_2.h>
#include <CGAL/Timer.h>
#include <list>
#include <fstream>

typedef CGAL::Exact_predicates_inexact_constructions_kernel   Kernel;
typedef Kernel::FT
   Number_type;
typedef CGAL::Arr_non_caching_segment_basic_traits_2<Kernel>  Traits_2;
typedef Traits_2::Point_2                                     Point_2;
typedef Traits_2::X_monotone_curve_2                          Segment_2;
typedef CGAL::Arrangement_2<Traits_2>
   Arrangement_2;

int main (int argc, char *argv[])
{
  // Get the name of the input file from the command line, or use the
   default
  // Europe.dat file if no command-line parameters are given.
  const char * filename = (argc > 1) ? argv[1] : "Europe.dat";

  // Open the input file.
  std::ifstream     in_file (filename);

  if (! in_file.is_open()) {
    std::cerr << "Failed to open " << filename << " ..." << std::endl;
    return (1);
  }

  // Read the segments from the file.
  // The input file format should be (all coordinate values are double
  // precision floating-point numbers):
  // <n>                                // number of segments.
  // <sx_1> <sy_1>  <tx_1> <ty_1>       // source and target of segment
   #1.
  // <sx_2> <sy_2>  <tx_2> <ty_2>       // source and target of segment
   #2.
  //   :       :       :       :
  // <sx_n> <sy_n>  <tx_n> <ty_n>       // source and target of segment
   #n.
  std::list<Segment_2>  segments;

  unsigned int n;
  in_file >> n;
  unsigned int i;
  for (i = 0; i < n; ++i) {
```

```
    double sx, sy, tx, ty;
    in_file >> sx >> sy >> tx >> ty;
    segments.push_back (Segment_2 (Point_2 (Number_type(sx),
   Number_type(sy)),
                                   Point_2 (Number_type(tx),
   Number_type(ty))));
  }
  in_file.close();

  // Construct the arrangement by aggregately inserting all segments.
  Arrangement_2                  arr;
  CGAL::Timer                    timer;

  std::cout << "Performing aggregated insertion of "
            << n << " segments." << std::endl;

  timer.start();
  insert_non_intersecting_curves (arr, segments.begin(), segments.end());
  timer.stop();

  // Print the arrangement dimensions.
  std::cout << "V = " << arr.number_of_vertices()
            << ",  E = " << arr.number_of_edges()
            << ",  F = " << arr.number_of_faces() << std::endl;

  std::cout << "Construction took " << timer.time()
            << " seconds." << std::endl;

  return 0;
}
```

File: examples/Arrangement_on_surface_2/predefined_kernel_non_intersecting.cpp

The *Arr_linear_traits_2*<*Kernel*> class used for demonstrating the construction of arrangements of unbounded curves is capable of handling bounded and unbounded linear objects, namely lines, rays and line segments. It is parameterized by a geometric kernel and such that its nested *Point_2* type is the same as the kernel point. The *Curve_2* (and *X_monotone_curve_2*) type it defines is constructible from a *Kernel::Line_2*, a *Kernel::Ray_2* or from a *Kernel::Segment_2* object. Just like the default segment-traits class, the linear-traits class also use caching techniques to speed up its predicates and constructions.

32.6.3 The Polyline-Traits Class

The *Arr_polyline_traits_2*<*SegmentTraits*> class can be used to maintain arrangements of polylines (a.k.a. poly-segments), which are continuous piecewise linear curves. A polyline can be created from a range of points, where the i-th and $(i+1)$-st points in the range represent the endpoints of the i-th segment of the polyline. The polyline traits class is parameterized with a segment-traits class that supports the basic operations on segments.

Polylines are the simplest form of a curves that are not necessarily x-monotone. They can be used to approximate more complicated curves in a convenient manner, as the algebra needed to handle them is elementary — rational arithmetic is sufficient to construct an arrangement of polylines is an exact and robust manner. Note, however,

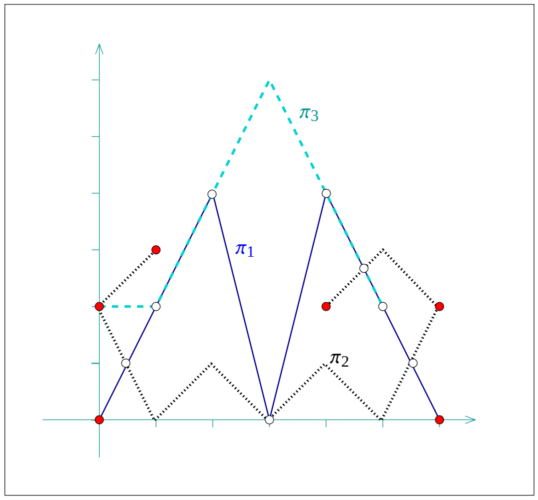


Figure 32.13: An arrangement of three polylines, as constructed in *polylines.cpp*. Disks mark vertices associated with polyline endpoints, while circles mark vertices that correspond to intersection points. Note that π_2 is split into three x-monotone polylines, and that π_1 and π_3 have two overlapping sections.

that a single polyline can be split into many x-monotone polylines, and that the number of intersection points (or overlapping sections) between two polylines can also be large.

The polyline-traits class is a model of the *ArrangementTraits_2* concept and of the *ArrangementLandmarkTraits_2* concept. It inherits its point type from the segment-traits class, and defines the polyline type, which serves as its *Curve_2*. Polyline curve objects can be constructed from a range of points. They also enable the traversal over the range of defining points, whose first and past-the-end iterators can be obtained through the methods *begin()* and *end()*. The nested *X_monotone_curve_2* type inherits from *Curve_2*. The points in an x-monotone curve are always stored in lexicographically increasing order of their coordinates.

The following example program constructs an arrangement of three polylines, as depicted in Figure 32.13. Note that most points defining the polylines are not associated with arrangement vertices. The arrangement vertices are either the extreme points of each x-monotone polyline or the intersection points between two polylines:

```
#include <CGAL/Cartesian.h>
#include <CGAL/Quotient.h>
#include <CGAL/MP_Float.h>
#include <CGAL/Arr_segment_traits_2.h>
#include <CGAL/Arr_polyline_traits_2.h>
#include <CGAL/Arrangement_2.h>
#include <vector>
#include <list>

#include "arr_print.h"

typedef CGAL::Quotient<CGAL::MP_Float>                      Number_type;
typedef CGAL::Cartesian<Number_type>                        Kernel;
typedef CGAL::Arr_segment_traits_2<Kernel>                  Segment_traits_2;
typedef CGAL::Arr_polyline_traits_2<Segment_traits_2>       Traits_2;
typedef Traits_2::Point_2                                   Point_2;
typedef Traits_2::Curve_2                                   Polyline_2;
```

```
typedef CGAL::Arrangement_2<Traits_2>                    Arrangement_2;

int main ()
{
  Arrangement_2         arr;

  Point_2               points1[5];
  points1[0] = Point_2 (0, 0);
  points1[1] = Point_2 (2, 4);
  points1[2] = Point_2 (3, 0);
  points1[3] = Point_2 (4, 4);
  points1[4] = Point_2 (6, 0);
  Polyline_2            pi1 (&points1[0], &points1[5]);

  std::list<Point_2>    points2;
  points2.push_back (Point_2 (1, 3));
  points2.push_back (Point_2 (0, 2));
  points2.push_back (Point_2 (1, 0));
  points2.push_back (Point_2 (2, 1));
  points2.push_back (Point_2 (3, 0));
  points2.push_back (Point_2 (4, 1));
  points2.push_back (Point_2 (5, 0));
  points2.push_back (Point_2 (6, 2));
  points2.push_back (Point_2 (5, 3));
  points2.push_back (Point_2 (4, 2));
  Polyline_2            pi2 (points2.begin(), points2.end());

  std::vector<Point_2>  points3 (4);
  points3[0] = Point_2 (0, 2);
  points3[1] = Point_2 (1, 2);
  points3[2] = Point_2 (3, 6);
  points3[3] = Point_2 (5, 2);
  Polyline_2            pi3 (points3.begin(), points3.end());

  insert (arr, pi1);
  insert (arr, pi2);
  insert (arr, pi3);

  print_arrangement (arr);
  return 0;
}
```

File: examples/Arrangement_on_surface_2/polylines.cpp

32.6.4 A Traits Class for Circular Arcs and Line Segments

Circles and circular arcs are the simplest form of non-linear curves. We handle circles whose centers have rational coordinates and whose squared radii is also rational. If we denote the circle center by (x_0, y_0) and its radius by r, then the equation of the circle — that is, $(x-x_0)^2 + (y-y_0)^2 = r^2$ — has rational coefficients. The intersection points of two such circles are therefore solutions of a quadratic equation with rational coefficients, or algebraic numbers of degree 2. The same applies for intersection points between such a rational circle and a

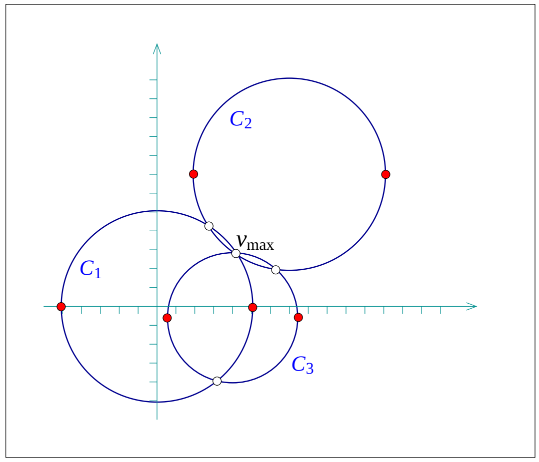


Figure 32.14: An arrangement of three circles constructed in *circles.cpp*. Each circle is split into two x-monotone circular arcs, whose endpoints are drawn as disks. Circles mark vertices that correspond to intersection points. The vertex $v_{\max}$ is a common intersection point of all three circles.

line, or a line segment, with rational coefficients (a line whose equation is $ax + by + c = 0$, where a, b and c are rational). Such numbers can be expressed as $\alpha + \beta\sqrt{\gamma}$, where α, β and γ are all rational numbers.

Arrangement of circular arcs and of line segment are very useful, as they occur in many applications. For example, when dilating a polygon by some radius we obtain a shape whose boundary is comprised of line segments, which correspond to dilated polygon edges, and circular arcs, which result from dilated polygon vertices. Using the arrangement of the boundary curves it is possible, for example, to compute the union of a set of dilated polygons.

The *Arr_circle_segment_traits_2<Kernel>* class-template is designed for efficient handling of arrangements of circular arcs and line segments. It is a model of the concepts *ArrangementTraits_2* and *ArrangementDirectionalXMonotoneTraits_2*; the later enables Boolean set operations. Note that it is not a model of *ArrangementLandmarkTraits_2* concept, so it is impossible to use the landmark point-location strategy. The traits class template is parameterized by a geometric kernel, and can handle arrangements of segments of *Kernel::Circle_2* objects (full circles are also supported) or of *Kernel::Line_2* objects—namely circular arcs and line segments. It is important to observe that the nested *Point_2* type defined by the traits class, whose coordinates are typically algebraic numbers of degree 2, is *not* the same as the *Kernel::Point_2* type, which is capable of representing a point with rational coordinates. The coordinates of a point are represented using the nested *CoordNT* number-type.

In the following example an arrangement of three full circles is constructed, as shown in Figure 32.14. Then, the vertex of maximal degree is searched for. The geometric mapping of this vertex is the point $(4,3)$, as all three circles intersect at this point and the associated vertex has six incident edges:

```
#include "arr_rational_nt.h"
#include <CGAL/Cartesian.h>
#include <CGAL/Arr_circle_segment_traits_2.h>
#include <CGAL/Arrangement_2.h>

typedef CGAL::Cartesian<Number_type>                    Kernel;
typedef Kernel::Circle_2                                Circle_2;
typedef CGAL::Arr_circle_segment_traits_2<Kernel>       Traits_2;
typedef Traits_2::CoordNT                               CoordNT;
```

```
typedef Traits_2::Point_2                                   Point_2;
typedef Traits_2::Curve_2                                   Curve_2;
typedef CGAL::Arrangement_2<Traits_2>                       Arrangement_2;

int main ()
{
  // Create a circle centered at the origin with radius 5.
  Kernel::Point_2  c1 = Kernel::Point_2 (0, 0);
  Number_type      sqr_r1 = Number_type (25);       // = 5^2
  Circle_2         circ1 = Circle_2 (c1, sqr_r1, CGAL::CLOCKWISE);
  Curve_2          cv1 = Curve_2 (circ1);

  // Create a circle centered at (7,7) with radius 5.
  Kernel::Point_2  c2 = Kernel::Point_2 (7, 7);
  Number_type      sqr_r2 = Number_type (25);       // = 5^2
  Circle_2         circ2 = Circle_2 (c2, sqr_r2, CGAL::CLOCKWISE);
  Curve_2          cv2 = Curve_2 (circ2);

  // Create a circle centered at (4,-0.5) with radius 3.5 (= 7/2).
  Kernel::Point_2  c3 = Kernel::Point_2 (4, Number_type (-1,2));
  Number_type      sqr_r3 = Number_type (49, 4);    // = 3.5^2
  Circle_2         circ3 = Circle_2 (c3, sqr_r3, CGAL::CLOCKWISE);
  Curve_2          cv3 = Curve_2 (circ3);

  // Construct the arrangement of the three circles.
  Arrangement_2    arr;

  insert (arr, cv1);
  insert (arr, cv2);
  insert (arr, cv3);

  // Locate the vertex with maximal degree.
  Arrangement_2::Vertex_const_iterator  vit;
  Arrangement_2::Vertex_const_handle    v_max;
  unsigned int                          max_degree = 0;

  for (vit = arr.vertices_begin(); vit != arr.vertices_end(); ++vit) {
    if (vit->degree() > max_degree) {
      v_max = vit;
      max_degree = vit->degree();
    }
  }

  std::cout << "The vertex with maximal degree in the arrangement is: "
            << "v_max = (" << v_max->point() << ") "
            << "with degree " << max_degree << "." << std::endl;
  return 0;
}
```

File: examples/Arrangement_on_surface_2/circles.cpp

The *Curve_2* type nested in *Arr_circle_segment_traits_2* can be used to represent circles, circular arcs, or line segments. Curve objects can therefore be constructed from a *Kernel::Circle_2* object or from a

Kernel::Segment_2 object. A circular arc is typically defined by a supporting circle and two endpoints, where the endpoints are instances of the *Point_2* type, with rational or irrational coordinates. The orientation of the arc is determined by the orientation of the supporting circle. Similarly, we also support the construction of lines segments given their supporting line (of type *Kernel::Line_2*) and two endpoints, which may have irrational coordinates (unlike the *Kernel::Segment_2* type).

Note that the *Kernel::Circle_2* type represents a circle whose *squared radius* is rational, where the radius itself may be irrational. However, if the radius is known to be rational, it is advisable to use it, for efficiency reasons. It is therefore also possible to construct a circle, or a circular arc specifying the circle center (a *Kernel::Point_2*), its rational radius, and its orientation. Finally, we also support the construction of a circular arcs that is defined by two endpoints and an arbitrary midpoint that lies on the arc in between its endpoint. In this case, all three points are required to have rational coordinates (to be kernel points).

The following example demonstrates the usage of the various construction methods for circular arcs and line segments. Note the usage of the constructor of *CoordNT (alpha, beta, gamma)*, which creates a degree-2 algebraic number whose value is $\alpha+\beta\sqrt{\gamma}$.

```
#include "arr_rational_nt.h"
#include <CGAL/Cartesian.h>
#include <CGAL/Arr_circle_segment_traits_2.h>
#include <CGAL/Arrangement_2.h>

typedef CGAL::Cartesian<Number_type>                     Kernel;
typedef Kernel::Circle_2                                 Circle_2;
typedef Kernel::Segment_2                                Segment_2;
typedef CGAL::Arr_circle_segment_traits_2<Kernel>        Traits_2;
typedef Traits_2::CoordNT                                CoordNT;
typedef Traits_2::Point_2                                Point_2;
typedef Traits_2::Curve_2                                Curve_2;
typedef CGAL::Arrangement_2<Traits_2>                    Arrangement_2;

int main ()
{
  std::list<Curve_2>  curves;

  // Create a circle centered at the origin with squared radius 2.
  Kernel::Point_2  c1 = Kernel::Point_2 (0, 0);
  Circle_2         circ1 = Circle_2 (c1, Number_type (2));

  curves.push_back (Curve_2 (circ1));

  // Create a circle centered at (2,3) with radius 3/2 - note that
  // as the radius is rational we use a different curve constructor.
  Kernel::Point_2  c2 = Kernel::Point_2 (2, 3);

  curves.push_back (Curve_2 (c2, Number_type(3, 2)));

  // Create a segment of the line (y = x) with rational endpoints.
  Kernel::Point_2  s3 = Kernel::Point_2 (-2, -2);
  Kernel::Point_2  t3 = Kernel::Point_2 (2, 2);
  Segment_2        seg3 = Segment_2 (s3, t3);

  curves.push_back (Curve_2 (seg3));
```

```
  // Create a line segment with the same supporting line (y = x), but
  // having one endpoint with irrational coefficients.
  CoordNT           sqrt_15 = CoordNT (0, 1, 15); // = sqrt(15)
  Point_2           s4 = Point_2 (3, 3);
  Point_2           t4 = Point_2 (sqrt_15, sqrt_15);

  curves.push_back (Curve_2 (seg3.supporting_line(), s4, t4));

  // Create a circular arc that correspond to the upper half of the
  // circle centered at (1,1) with squared radius 3. We create the
  // circle with clockwise orientation, so the arc is directed from
  // (1 - sqrt(3), 1) to (1 + sqrt(3), 1).
  Kernel::Point_2  c5 = Kernel::Point_2 (1, 1);
  Circle_2          circ5 = Circle_2 (c5, 3, CGAL::CLOCKWISE);
  CoordNT           one_minus_sqrt_3 = CoordNT (1, -1, 3);
  CoordNT           one_plus_sqrt_3 = CoordNT (1, 1, 3);
  Point_2           s5 = Point_2 (one_minus_sqrt_3, CoordNT (1));
  Point_2           t5 = Point_2 (one_plus_sqrt_3, CoordNT (1));

  curves.push_back (Curve_2 (circ5, s5, t5));

  // Create a circular arc of the unit circle, directed clockwise from
  // (-1/2, sqrt(3)/2) to (1/2, sqrt(3)/2). Note that we orient the
  // supporting circle accordingly.
  Kernel::Point_2  c6 = Kernel::Point_2 (0, 0);
  CoordNT           sqrt_3_div_2 = CoordNT (Number_type(0),
   Number_type(1,2), Number_type(3));
  Point_2           s6 = Point_2 (Number_type (-1, 2), sqrt_3_div_2);
  Point_2           t6 = Point_2 (Number_type (1, 2), sqrt_3_div_2);

  curves.push_back (Curve_2 (c6, 1, CGAL::CLOCKWISE, s6, t6));

  // Create a circular arc defined by two endpoints and a midpoint,
  // all having rational coordinates. This arc is the upper-right
  // quarter of a circle centered at the origin with radius 5.
  Kernel::Point_2  s7 = Kernel::Point_2 (0, 5);
  Kernel::Point_2  mid7 = Kernel::Point_2 (3, 4);
  Kernel::Point_2  t7 = Kernel::Point_2 (5, 0);

  curves.push_back (Curve_2 (s7, mid7, t7));

  // Construct the arrangement of the curves.
  Arrangement_2     arr;

  insert (arr, curves.begin(), curves.end());

  // Print the size of the arrangement.
  std::cout << "The arrangement size:" << std::endl
            << "   V = " << arr.number_of_vertices()
            << ",  E = " << arr.number_of_edges()
            << ",  F = " << arr.number_of_faces() << std::endl;

  return 0;
}
```

File: `examples/Arrangement_on_surface_2/circular_arcs.cpp`

It is also possible to construct x-monotone curve objects, which represent x-monotone circular arcs or line segments, using similar constructors. Construction from a full circle is obviously not supported. See the Reference Manual for more details.

The traits class-template *Arr_circular_line_arc_traits_2<CircularKernel>* offered by the arrangement package also handles circular arcs and line segments. It is an alternative to the *Arr_circle_segment_traits_2<Kernel>* class-template. These two class templates, while serve similar purposes, are based on different concepts, and posses different characteristics. You are encouraged to experiment with both, compare their performance, and use the most suitable for your case.

32.6.5 A Traits Class for Conic Arcs

A *conic curve* is an algebraic curve of degree 2. Namely, it is the locus of all points (x,y) satisfying the equation $C:\ rx^2+sy^2+txy+ux+vy+w=0$, where the six coefficients $\langle r,s,t,u,v,w\rangle$ completely characterize the curve. The sign of the expression $\Delta_C = 4rs-t^2$ determines the type of curve:

- If $\Delta_C > 0$ the curve is an ellipse. A circle is a special case of an ellipse, where $r=s$ and $t=0$.
- If $\Delta_C = 0$ the curve is a parabola — an unbounded conic curve with a single connected branch. When $r=s=t=0$ we have a line, which can be considered as a degenerate parabola.
- If $\Delta_C < 0$ the curve is a hyperbola. That is, it is comprised of two disconnected unbounded branches.

As the arrangement package is suitable for bounded curves, we consider bounded segments of conic curves, referred to as *conic arcs*. A conic arc a may be either (i) a full ellipse, or (ii) defined by the tuple $\langle C,p_s,p_t,o\rangle$, where C is a conic curve and p_s and p_t are two points on C (namely $C(p_s)=C(p_t)=0$) that define the *source* and *target* of the arc, respectively. The arc is formed by traversing C from the source to the target going in the orientation specified by o, which is typically clockwise or counterclockwise orientation (but may also be collinear in case of degenerate conic curves).

We always assume that the conic coefficients $\langle r,s,t,u,v,w\rangle$ are rational. When dealing with linear curves (line segments and polylines), similar assumptions guarantee that all intersection points also have rational coordinates, such that the arrangement of such curves can be constructed and maintained using only rational arithmetic. Unfortunately, this does not hold for conic curves, as the coordinates of intersection points of two conic curves with rational coefficients are in general algebraic numbers of degree 4.[17] In addition, conic arcs may not necessarily be x-monotone, and must be split at points where the tangent to the arc is vertical. In the general case, such points typically have coordinates that are algebraic numbers of degree 2. It is therefore clear that we have to use different number types to represent the conic coefficients and the point coordinates. Note that as arrangement vertices induced by intersection points and points with vertical tangents are likely to have algebraic coordinates, we also allow the original endpoints of the input arcs p_s and p_t to have algebraic coordinates.

The *Arr_conic_traits_2<RatKernel, AlgKernel, NtTraits>* class template is designed for efficient handling of arrangements of bounded conic arcs. The template has three parameters, defined as follows:

- The *RatKernel* class is a geometric kernel, whose field type is an exact rational type. It is used to define basic geometric entities (e.g., a line segment or a circle) with rational coefficients. Typically we use one of the standard CGAL kernels, instantiated with the number type *NtTraits::Rational* (see below).

[17] Namely, they are roots of polynomials with integer coefficients of degree 4. However, in some special cases, for example when handling only circles and circular arcs, the coordinates of the intersection points are only of degree 2, namely they are solutions of quadratic equations.

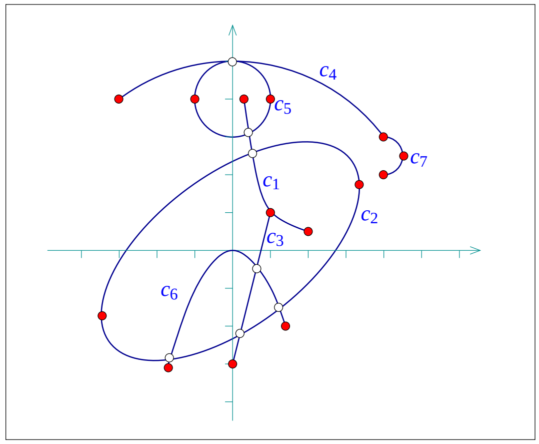


Figure 32.15: An arrangement of mixed conic arcs, as constructed in *conics.cpp*.

- The *AlgKernel* class is a geometric kernel whose field type is an exact algebraic type. It is used to define points with algebraic coordinates. Typically we use one of the standard CGAL kernels, instantiated with the number type *NtTraits::Algebraic* (see below).
- The *NtTraits* class (the number-type traits class) encapsulates all the numeric operations needed for performing the geometric computation carried out by the geometric traits class. It defines the *Integer*, *Rational* and *Algebraic* number-types, and supports several operations on these types, such as conversion between number types, solving quadratic equations and extracting the real roots of a polynomial with integer coefficients. It is highly recommended to use the *CORE_algebraic_number_traits* class, which is included in the arrangement package. It relies on the exact number types implemented in the CORE library and performs exact computations on the number types it defines.

The *Arr_conic_traits_2* models the *ArrangementTraits_2* and *ArrangementLandmarkTraits_2* concepts. (It supports the landmark point-location strategy). Its *Point_2* type is derived from *AlgKernel::Point_2*, while the *Curve_2* type represents a bounded, not necessarily x-monotone, conic arc. The *X_monotone_curve_2* type is derived from *Curve_2*, but its constructors are to be used only by the traits class. You should therefore construct only *Curve_2* objects and insert them into the arrangement using the *insert()* or *insert()* functions.

Conic arcs can be constructed from full ellipses or by specifying a supporting curve, two endpoints and an orientation. However, several constructors of *Curve_2* are available to allow for some special cases, such as circular arcs or line segments. The *Curve_2* (and the derived *X_monotone_curve_2*) classes also support basic access functions such as *source()*, *target()* and *orientation()*.

Examples for Arrangements of Conics

The following example demonstrates the usage of the various constructors for conic arcs. The resulting arrangement is depicted in Figure 32.15. Especially noteworthy are the constructor of a circular arc that accepts three points and the constructor that allows specifying approximate endpoints, where the exact endpoints are given explicitly as intersections of the supporting conic with two other conic curves. Also note that as the preconditions required by some of these constructors are rather complicated (see the Reference Manual for the details), a precondition violation does not cause the program to terminate — instead, an *invalid* arc is created. We can verify the validity of an arc by using the *is_valid()* method. Needless to say, inserting invalid arcs into an arrangement is not allowed.

```
#include <CGAL/basic.h>
```

```
#ifndef CGAL_USE_CORE
#include <iostream>
int main ()
{
  std::cout << "Sorry, this example needs CORE ..." << std::endl;
  return 0;
}
#else

#include <CGAL/Cartesian.h>
#include <CGAL/CORE_algebraic_number_traits.h>
#include <CGAL/Arr_conic_traits_2.h>
#include <CGAL/Arrangement_2.h>

typedef CGAL::CORE_algebraic_number_traits              Nt_traits;
typedef Nt_traits::Rational                             Rational;
typedef Nt_traits::Algebraic                            Algebraic;
typedef CGAL::Cartesian<Rational>                       Rat_kernel;
typedef Rat_kernel::Point_2                             Rat_point_2;
typedef Rat_kernel::Segment_2                           Rat_segment_2;
typedef Rat_kernel::Circle_2                            Rat_circle_2;
typedef CGAL::Cartesian<Algebraic>                      Alg_kernel;
typedef CGAL::Arr_conic_traits_2<Rat_kernel, Alg_kernel, Nt_traits>
                                                        Traits_2;
typedef Traits_2::Point_2                               Point_2;
typedef Traits_2::Curve_2                               Conic_arc_2;
typedef CGAL::Arrangement_2<Traits_2>                   Arrangement_2;

int main ()
{
  Arrangement_2    arr;

  // Insert a hyperbolic arc, supported by the hyperbola y = 1/x
  // (or: xy - 1 = 0) with the endpoints (1/5, 4) and (2, 1/2).
  // Note that the arc is counterclockwise oriented.
  Point_2       ps1 (Rational(1,4), 4);
  Point_2       pt1 (2, Rational(1,2));
  Conic_arc_2   c1 (0, 0, 1, 0, 0, -1, CGAL::COUNTERCLOCKWISE, ps1, pt1);

  insert (arr, c1);

  // Insert a full ellipse, which is (x/4)^2 + (y/2)^2 = 0 rotated by
  // phi=36.87 degree (such that sin(phi) = 0.6, cos(phi) = 0.8),
  // yielding: 58x^2 + 72y^2 - 48xy - 360 = 0.
  Conic_arc_2   c2 (58, 72, -48, 0, 0, -360);

  insert (arr, c2);

  // Insert the segment (1, 1) -- (0, -3).
  Rat_point_2   ps3 (1, 1);
  Rat_point_2   pt3 (0, -3);
  Conic_arc_2   c3 (Rat_segment_2 (ps3, pt3));
```

```
insert (arr, c3);

// Insert a circular arc supported by the circle x^2 + y^2 = 5^2,
// with (-3, 4) and (4, 3) as its endpoints. We want the arc to be
// clockwise oriented, so it passes through (0, 5) as well.
Rat_point_2   ps4 (-3, 4);
Rat_point_2   pm4 (0, 5);
Rat_point_2   pt4 (4, 3);
Conic_arc_2   c4 (ps4, pm4, pt4);

CGAL_assertion (c4.is_valid());
insert (arr, c4);

// Insert a full unit circle that is centered at (0, 4).
Rat_circle_2  circ5 (Rat_point_2(0,4), 1);
Conic_arc_2   c5 (circ5);

insert (arr, c5);

// Insert a parabolic arc that is supported by a parabola y = -x^2
// (or: x^2 + y = 0) and whose endpoints are (-sqrt(3), -3) ~ (-1.73,
 -3)
// and (sqrt(2), -2) ~ (1.41, -2). Notice that since the x-coordinates
// of the endpoints cannot be acccurately represented, we specify them
// as the intersections of the parabola with the lines y = -3 and y =
 -2.
// Note that the arc is clockwise oriented.
Conic_arc_2   c6 =
  Conic_arc_2 (1, 0, 0, 0, 1, 0,       // The parabola.
               CGAL::CLOCKWISE,
               Point_2 (-1.73, -3),    // Approximation of the source.
               0, 0, 0, 0, 1, 3,       // The line: y = -3.
               Point_2 (1.41, -2),     // Approximation of the target.
               0, 0, 0, 0, 1, 2);      // The line: y = -2.

CGAL_assertion (c6.is_valid());
insert (arr, c6);

// Insert the right half of the circle centered at (4, 2.5) whose
 radius
// is 1/2 (therefore its squared radius is 1/4).
Rat_circle_2  circ7 (Rat_point_2(4, Rational(5,2)), Rational(1,4));
Point_2       ps7 (4, 3);
Point_2       pt7 (4, 2);
Conic_arc_2   c7 (circ7, CGAL::CLOCKWISE, ps7, pt7);

insert (arr, c7);

// Print out the size of the resulting arrangement.
std::cout << "The arrangement size:" << std::endl
          << "   V = " << arr.number_of_vertices()
          << ",  E = " << arr.number_of_edges()
          << ",  F = " << arr.number_of_faces() << std::endl;
```

```
  return 0;
}

#endif
```

File: examples/Arrangement_on_surface_2/conics.cpp

The last example in this section demonstrates how the conic-traits class can handle intersection points with multiplicity. The supporting curves of the two arcs, a circle centered at $(0, \frac{1}{2})$ with radius $\frac{1}{2}$, and the hyperbola $y = \frac{x^2}{1-x}$,[18] intersect at the origin such that the intersection point has multiplicity 3 (note that they both have the same horizontal tangent at $(0,0)$ and the same curvature 1). In addition, they have another intersection point at $(\frac{1}{2}, \frac{1}{2})$ of multiplicity 1:

```
#include <CGAL/basic.h>

#ifndef CGAL_USE_CORE
#include <iostream>
int main ()
{
  std::cout << "Sorry, this example needs CORE ..." << std::endl;
  return 0;
}
#else

#include <CGAL/Cartesian.h>
#include <CGAL/CORE_algebraic_number_traits.h>
#include <CGAL/Arr_conic_traits_2.h>
#include <CGAL/Arrangement_2.h>
#include <CGAL/Arr_naive_point_location.h>

#include "arr_print.h"

typedef CGAL::CORE_algebraic_number_traits            Nt_traits;
typedef Nt_traits::Rational                           Rational;
typedef Nt_traits::Algebraic                          Algebraic;
typedef CGAL::Cartesian<Rational>                     Rat_kernel;
typedef Rat_kernel::Point_2                           Rat_point_2;
typedef Rat_kernel::Segment_2                         Rat_segment_2;
typedef Rat_kernel::Circle_2                          Rat_circle_2;
typedef CGAL::Cartesian<Algebraic>                    Alg_kernel;
typedef CGAL::Arr_conic_traits_2<Rat_kernel,
                                 Alg_kernel,
                                 Nt_traits>           Traits_2;
typedef Traits_2::Point_2                             Point_2;
typedef Traits_2::Curve_2                             Conic_arc_2;
typedef CGAL::Arrangement_2<Traits_2>                 Arrangement_2;
typedef CGAL::Arr_naive_point_location<Arrangement_2> Naive_pl;

int main ()
{
```

[18] This curve can also be written as $C : x^2 + xy - y = 0$. It is a hyperbola since $\Delta_C = -1$.

```
  Arrangement_2  arr;
  Naive_pl       pl (arr);

  // Insert a hyperbolic arc, supported by the hyperbola y = x^2/(1-x)
  // (or: x^2 + xy - y = 0) with the endpoints (-1, 1/2) and (1/2, 1/2).
  // Note that the arc is counterclockwise oriented.
  Point_2        ps1 (-1, Rational(1,2));
  Point_2        pt1 (Rational(1,2), Rational(1,2));
  Conic_arc_2    cv1 (1, 0, 1, 0, -1, 0, CGAL::COUNTERCLOCKWISE, ps1,
   pt1);

  insert (arr, cv1, pl);

  // Insert the bottom half of the circle centered at (0, 1/2) whose
   radius
  // is 1/2 (therefore its squared radius is 1/4).
  Rat_circle_2   circ2 (Rat_point_2(0, Rational(1,2)), Rational(1,4));
  Point_2        ps2 (-Rational(1,2), Rational(1,2));
  Point_2        pt2 (Rational(1,2), Rational(1,2));
  Conic_arc_2    cv2 (circ2, CGAL::COUNTERCLOCKWISE, ps2, pt2);

  insert (arr, cv2, pl);

  // Print the resulting arrangement.
  print_arrangement (arr);

  return 0;
}

#endif
```

File: examples/Arrangement_on_surface_2/conic_multiplicities.cpp

32.6.6 A Traits Class for Arcs of Rational Functions

The traits class *Arr_rational_function_traits_2<AlgebraicKernel_d_1>* handles bounded and unbounded arcs of rational functions, referred to as *rational arcs* (in particular, such an arc may correspond to the entire graph of a rational function), and enables the construction and maintenance of arrangements of such arcs. Rational functions, and polynomial functions in particular, are not only interesting in their own right, they are also very useful for approximating or interpolating more complicated curves; see, e.g., [PTVF02, Chapter 3].

Arr_rational_function_traits_2<AlgebraicKernel_d_1> is a model of the concepts *ArrangementTraits_2*, *ArrangementOpenBoundaryTraits_2*, and *ArrangementDirectionalXMonotoneTraits_2*; the later enables Boolean set operations. Note that it is not a model of *ArrangementLandmarkTraits_2* concept, so it is impossible to use the landmark point-location strategy with this traits class.

A rational function $y = \frac{P(x)}{Q(x)}$ is defined by two polynomials P and Q of arbitrary degrees. If $Q(x) = 1$ then the function is a simple polynomial function. Usually the domain is $\mathbb{R}$ but the function may also be restricted to a bounded interval $[x_{\min}, x_{\max}]$ or defined over a ray $(-\infty, x_{\max}]$ or $[x_{\min}, \infty)$. Rational functions are represented by the nested type *Curve_2*. A rational arc is always x-monotone in the mathematical sense. However, it is not necessarily continuous, as it may have singularities. An arc that has singularities must be split into continuous portions before being inserted into the arrangement. Arbitrary rational functions are represented by the nested type *Curve_2* and continuous portions of rational functions are represented by the nested type *X_monotone_curve_2*. Constructors for both types are provided by the traits. A *Curve_2* may be split up into several *X_monotone_curve_2* using *Make_x_monotone_2*.

Using the *Arr_rational_function_traits_2<AlgebraicKernel_d_1>* class template it is possible to construct and maintain arrangement of rational arcs. The template parameter of the traits must be a model of the concept *AlgebraicKernel_d_1*. A rational function is represented as the quotient of two polynomials P and Q of type *AlgebraicKernel_d_1::Polynomial_1* and an x-interval over which the polynomials are defined. The type of the polynomial coefficients, namely *AlgebraicKernel_d_1::Coefficient*, cannot be algebraic. Moreover, it is recommended that this type is not made rational either, since using rational, as opposed to integral, coefficients does not extend the range of the rational arcs and is typically less efficient. The type of the interval bounds, namely *AlgebraicKernel_d_1::Bound*, however, can be algebraic. A point is represented by a rational function and its x-coordinate, which is of type *AlgebraicKernel_d_1::Algebraic_real_1*. Note that an explicit representation of the y-coordinate is only computed upon request, as it can be a rather costly operation.

The constructed rational functions are cached by the traits class. The cache is local to each traits class object. It is therefore necessary to construct curves using only the constructor objects provided by member functions of the traits class. Moreover, a curve must only be used by the traits class object that was used to construct it. The cache is automatically cleaned up from time to time. The amortized clean up costs are constant. In addition, there is also a separate member function that cleans up the cache on demand.

The curve constructors have an additional advantage. They conveniently enable the provision of two polynomials that define a rational arc using rational coefficients. For example, let P and Q denote two polynomials with integral coefficients that define a rational arc at interest, and let P' and Q' denote two polynomials with rational coefficients that define the same rational arc; that is, the quotients P/Q and P'/Q' are identical. You can construct the rational arc providing the coefficients of P' and Q' to the constructor. In this case the constructor normalizes the coefficients and stores the desired polynomials P and Q.

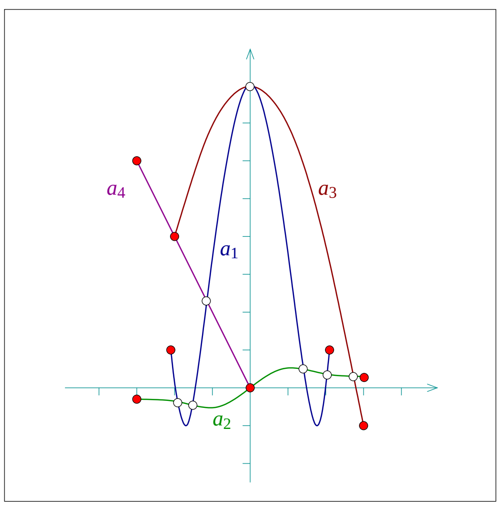


Figure 32.16: An arrangement of four arcs of rational functions, as constructed in *rational_functions.cpp*.

The following example demonstrates the construction of an arrangement of rational arcs depicted in Figure 32.16. Note the usage of the two constructors, for polynomial arcs and for rational arcs:

```
#include <CGAL/basic.h>

#ifndef CGAL_USE_CORE
```

```
#include <iostream>
int main ()
{
  std::cout << "Sorry, this example needs CORE ..." << std::endl;
  return 0;
}

#else

#include <CGAL/CORE_BigInt.h>                        // NT
#include <CGAL/Algebraic_kernel_d_1.h>               // Algebraic Kernel
#include <CGAL/Arr_rational_function_traits_2.h>     // Traits
#include <CGAL/Arrangement_2.h>                      // Arrangement

typedef CORE::BigInt                                 Number_type;
typedef CGAL::Algebraic_kernel_d_1<Number_type>      AK1;
typedef CGAL::Arr_rational_function_traits_2<AK1>    Traits_2;

typedef Traits_2::Polynomial_1                       Polynomial_1;
typedef Traits_2::Algebraic_real_1                   Alg_real_1;

typedef CGAL::Arrangement_2<Traits_2>                Arrangement_2;

int main ()
{
  CGAL::set_pretty_mode(std::cout);               // for nice printouts.

  // create a polynomial representing x .-)
  Polynomial_1 x = CGAL::shift(Polynomial_1(1),1);

  // Traits class object
  Traits_2 traits;
  Traits_2::Construct_x_monotone_curve_2 construct_arc
    = traits.construct_x_monotone_curve_2_object();

  // container storing all arcs
  std::vector<Traits_2::X_monotone_curve_2>  arcs;

  // Create an arc supported by the polynomial y = x^4 - 6x^2 + 8,
  // defined over the interval [-2.1, 2.1]:
  Polynomial_1 P1 = x*x*x*x - 6*x*x + 8;
  Alg_real_1 l(Traits_2::Algebraic_kernel_d_1::Bound(-2.1));
  Alg_real_1 r(Traits_2::Algebraic_kernel_d_1::Bound(2.1));
  arcs.push_back(construct_arc(P1, l, r));

  // Create an arc supported by the function y = x / (1 + x^2),
  // defined over the interval [-3, 3]:
  Polynomial_1 P2 = x;
  Polynomial_1 Q2 = 1+x*x;

  arcs.push_back(construct_arc(P2, Q2, Alg_real_1(-3), Alg_real_1(3)));

  // Create an arc supported by the parbola y = 8 - x^2,
  // defined over the interval [-2, 3]:
```

```
  Polynomial_1 P3 = 8 - x*x;
  arcs.push_back(construct_arc(P3, Alg_real_1(-2), Alg_real_1(3)));

  // Create an arc supported by the line y = -2x,
  // defined over the interval [-3, 0]:
  Polynomial_1 P4 = -2*x;
  arcs.push_back(construct_arc(P4, Alg_real_1(-3), Alg_real_1(0)));

  // Construct the arrangement of the four arcs.

  // Print the arcs.
  for (unsigned int i(0); i < arcs.size(); ++i)
    std::cout << arcs[i]<<std::endl;


  Arrangement_2 arr(&traits);
  insert(arr, arcs.begin(), arcs.end());

  // Print the arrangement size.
  std::cout << "The arrangement size:" << std::endl
            << "   V = " << arr.number_of_vertices()
            << ",  E = " << arr.number_of_edges()
            << ",  F = " << arr.number_of_faces() << std::endl;

  return 0;
}

#endif
```

File: examples/Arrangement_on_surface_2/rational_functions.cpp

The following example demonstrates the construction of an arrangement of six rational arcs—four unbounded arcs and two bounded ones—as depicted in Figure 32.17. Note the usage of the constructors of an entire rational function and of an infinite "ray" of such a function. Also observe that the hyperbolas $y = \pm\frac{1}{x}$ and $y = \pm\frac{1}{2x}$ never intersect, although they have common vertical and horizontal asymptotes, so very "thin" unbounded faces are created between them:

```
#include <CGAL/basic.h>

#ifndef CGAL_USE_CORE
#include <iostream>
int main ()
{
  std::cout << "Sorry, this example needs CORE ..." << std::endl;
  return 0;
}

#else

#include <CGAL/CORE_BigInt.h>                      // NT
#include <CGAL/Algebraic_kernel_d_1.h>             // Algebraic Kernel
#include <CGAL/Arr_rational_function_traits_2.h> // Traits
```

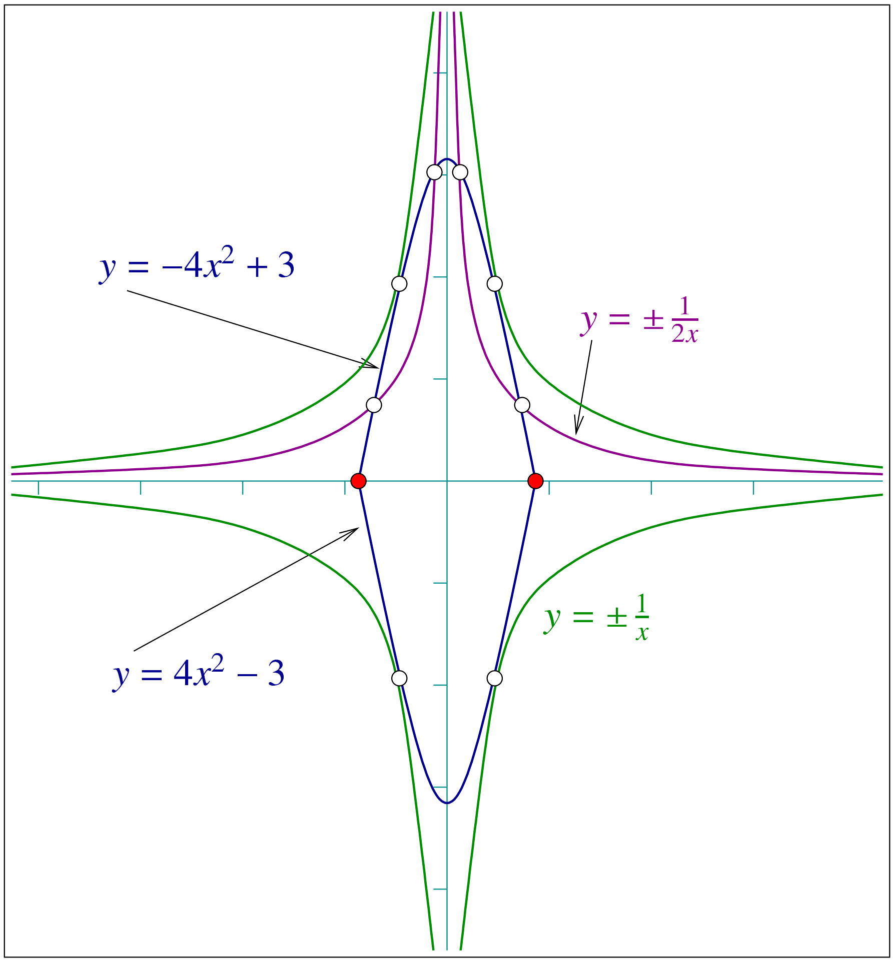


Figure 32.17: An arrangement of six arcs of rational functions, as constructed in *unbounded_rational_functions.cpp*.

```
#include <CGAL/Arrangement_2.h>                          // Arrangement

typedef CORE::BigInt                                       Number_type;
typedef CGAL::Algebraic_kernel_d_1<Number_type>            AK1;
typedef CGAL::Arr_rational_function_traits_2<AK1>          Traits_2;

typedef Traits_2::Polynomial_1                             Polynomial_1;
typedef Traits_2::Algebraic_real_1                         Alg_real_1;

typedef CGAL::Arrangement_2<Traits_2>                      Arrangement_2;

int main ()
{
  CGAL::set_pretty_mode(std::cout);                       // for nice printouts.

  // Traits class object
  AK1 ak1;
  Traits_2 traits(&ak1);
```

```
// constructor for rational functions
Traits_2::Construct_curve_2 construct =
  traits.construct_curve_2_object();

// a polynomial representing x .-)
Polynomial_1 x = CGAL::shift(Polynomial_1(1),1);

// container storing all arcs
std::vector<Traits_2::Curve_2>  arcs;


// Create the rational functions (y = 1 / x), and (y = -1 / x).
Polynomial_1 P1(1);
Polynomial_1 minusP1(-P1);
Polynomial_1 Q1 = x;
arcs.push_back(construct(P1, Q1));
arcs.push_back(construct(minusP1, Q1));

// Create a bounded segments of the parabolas (y = -4*x^2 + 3) and
// (y = 4*x^2 - 3), defined over [-sqrt(3)/2, sqrt(3)/2].
Polynomial_1 P2 = -4*x*x+3;
Polynomial_1 minusP2 = -P2;
std::vector<std::pair<Alg_real_1,int> > roots;

// [-sqrt(3)/2, sqrt(3)/2]
traits.algebraic_kernel_d_1()->solve_1_object()(P2,
  std::back_inserter(roots));
arcs.push_back(construct(P2, roots[0].first, roots[1].first));
arcs.push_back(construct(minusP2, roots[0].first, roots[1].first));

// Create the rational function (y = 1 / 2*x) for x > 0, and the
// rational function (y = -1 / 2*x) for x < 0.
Polynomial_1 P3(1);
Polynomial_1 minusP3(-P3);
Polynomial_1 Q3 = 2*x;
arcs.push_back(construct(P3, Q3, Alg_real_1(0), true));
arcs.push_back(construct(minusP3, Q3, Alg_real_1(0), false));

// Construct the arrangement of the six arcs.
//Arrangement_2 arr(&traits);
Arrangement_2 arr;
insert(arr, arcs.begin(), arcs.end());

// Print the arrangement size.
std::cout << "The arrangement size:" << std::endl
   << "   V = " << arr.number_of_vertices()
   << " (plus " << arr.number_of_vertices_at_infinity()
   << " at infinity)"
   << ",  E = " << arr.number_of_edges()
   << ",  F = " << arr.number_of_faces()
   << " (" << arr.number_of_unbounded_faces() << " unbounded)"
   << std::endl << std::endl;
```

```
  return 0;
}

#endif
```

File: examples/Arrangement_on_surface_2/unbounded_rational_functions.cpp

32.6.7 A Traits Class for Planar Bézier Curves

A planar *Bézier curve B* is a parametric curve defined by a sequence of *control points* $p_0, \ldots, p_n$ as follows:

$$B(t) = (X(t), Y(t)) = \sum_{k=0}^{n} p_k \cdot \frac{n!}{k!(n-k)!} \cdot t^k (1-t)^{n-k} \,.$$

where $t \in [0,1]$. The degree of the curve is therefore n — namely, $X(t)$ and $Y(t)$ are polynomials of degree n. Bézier curves have numerous applications in computer graphics and solid modelling. They are used, for example, in free-form sketches and for defining the true-type fonts.

Using the *Arr_Bezier_curve_traits_2<RatKernel, AlgKernel, NtTraits>* class template it is possible to construct and maintain arrangements of Bézier curves that are given by rational control points (a sequence of objects of the *RatKernel::Point_2* type). We can handle curves of arbitrary degree (in general, a sequence of $n+1$ control points define a Bézier curve of degree n). The template parameters are the same ones used by the *Arr_conic_traits_2* class template, and here it is also recommended to use the *CORE_algebraic_number_traits* class, with Cartesian kernels instantiated with the *Rational* and *Algebraic* number-types defined by this class.

As mentioned above, we assume that the coordinates of all control points that define a Bézier curve are rational numbers, so both $X(t)$ and $Y(t)$ are polynomials with rational coefficients. The intersection points between curves are however algebraic numbers, and their exact computation is time-consuming. The traits class therefore contains a layer of geometric filtering that performs all computation in an approximate manner whenever possible. Thus, it resorts to exact computations only when the approximate computation fails to produce an unambiguous result. Note that most arrangement vertices are therefore associated with approximated points. You cannot access the coordinates of such points and obtain them as algebraic numbers, and only access to the approximate coordinates in possible. See the Reference Manual for the exact interface of the *Point_2*, *Curve_2* and *X_monotone_curve_2* defined by the traits class.

The *Arr_Bezier_curve_traits_2* is a model of the *ArrangementTraits_2* concept (but not of the *ArrangementLandmarkTraits_2* concept, so it is impossible to use the landmark point-location strategy for arrangements of rational arcs).

The following example reads a set of Bézier curves from an input file, where each file is specified by an integer stating its number of control points, followed by the sequence of control points, given in integer or rational coordinates. By default, the program uses the *Bezier.dat* file, which contains ten curves of degree 5 each; their resulting arrangement is depicted in Figure 32.18.

```
#include <CGAL/basic.h>

#ifndef CGAL_USE_CORE
#include <iostream>
int main ()
{
  std::cout << "Sorry, this example needs CORE ..." << std::endl;
```

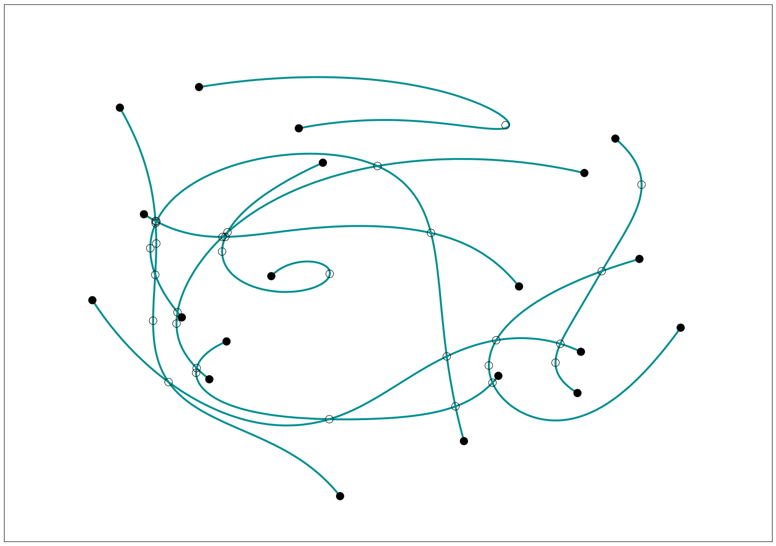


Figure 32.18: An arrangement of ten Bézier curves of degree 5, as constructed in *Bezier_curves.cpp*.

```
  return 0;
}
#else

#include <CGAL/Cartesian.h>
#include <CGAL/CORE_algebraic_number_traits.h>
#include <CGAL/Arr_Bezier_curve_traits_2.h>
#include <CGAL/Arrangement_2.h>

typedef CGAL::CORE_algebraic_number_traits              Nt_traits;
typedef Nt_traits::Rational                             NT;
typedef Nt_traits::Rational                             Rational;
typedef Nt_traits::Algebraic                            Algebraic;
typedef CGAL::Cartesian<Rational>                       Rat_kernel;
typedef CGAL::Cartesian<Algebraic>                      Alg_kernel;
typedef Rat_kernel::Point_2                             Rat_point_2;
typedef CGAL::Arr_Bezier_curve_traits_2<Rat_kernel, Alg_kernel,
   Nt_traits>
                                                        Traits_2;
typedef Traits_2::Curve_2                               Bezier_curve_2;
typedef CGAL::Arrangement_2<Traits_2>                   Arrangement_2;

int main (int argc, char *argv[])
{
  // Get the name of the input file from the command line, or use the
   default
  // Bezier.dat file if no command-line parameters are given.
  const char   *filename = (argc > 1) ? argv[1] : "Bezier.dat";

  // Open the input file.
  std::ifstream   in_file (filename);
```

```
  if (! in_file.is_open()) {
    std::cerr << "Failed to open " << filename << std::endl;
    return 1;
  }

  // Read the curves from the input file.
  unsigned int              n_curves;
  std::list<Bezier_curve_2>  curves;
  Bezier_curve_2             B;
  unsigned int              k;

  in_file >> n_curves;
  for (k = 0; k < n_curves; k++) {
    // Read the current curve (specified by its control points).
    in_file >> B;
    curves.push_back (B);

    std::cout << "B = {" << B << "}" << std::endl;
  }

  // Construct the arrangement.
  Arrangement_2                     arr;
  insert (arr, curves.begin(), curves.end());

  // Print the arrangement size.
  std::cout << "The arrangement size:" << std::endl
            << "   V = " << arr.number_of_vertices()
            << ",  E = " << arr.number_of_edges()
            << ",  F = " << arr.number_of_faces() << std::endl;

  return 0;
}

#endif
```

File: examples/Arrangement_on_surface_2/Bezier_curves.cpp

32.6.8 A Traits Class for Planar Algebraic Curves of Arbitrary Degree

An algebraic curve C in the plane is defined as the (real) zero locus of a polynomial $f(x,y)$ in two variables. The curve is uniquely defined by f (although several polynomials might define the same curve). We call f a *defining polynomial* of C.

We consider arrangements induced by algebraic curves or by (weakly) x-monotone segments for algebraic curves (Such a segment is not necessarily the maximal possible (weakly) x-monotone segment; see below.) When talking about algebraic curves, we use the term "segment" for a continuous, possibly non-linear subset of an algebraic curve – see the definition below. There are no restrictions on the algebraic curve, that means, we support unbounded curves, vertical curves or segments, and isolated points.

The *Arr_algebraic_segment_traits_2<Coefficient>* class template is a model of the *ArrangementTraits_2* concept (but not of the *ArrangementLandmarkTraits_2* concept, so it is impossible to use the landmark point-location

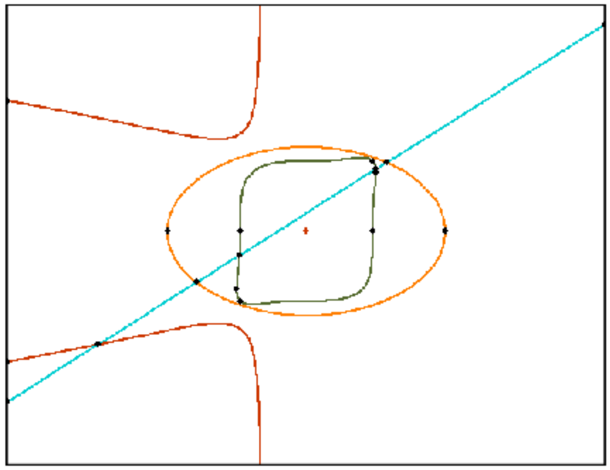


Figure 32.19: An arrangement of algebraic curves of degrees 1, 2, 3, and 6, as constructed in *algebraic_curves.cpp*.

strategy for arrangements of algebraic curves). The template argument *Coefficient* determines the type of the scalar coefficients of the polynomial. Currently supported types are *leda_integer*, *CORE::BigInt*, and any instance of *CGAL::Sqrt_extension<A,B>* instantiated with one of the integral types above.

The traits class defines a type *Curve_2* for algebraic curves. Such a type can be constructed by the *Construct_curve_2* functor, which accepts an instance of *Polynomial_2* as an argument. This polynomial type is also available by the traits class and constitutes a valid model of the concept *Polynomial_d* with two variables (see ??).

The following examples computes the arrangement induced by the four curves in Figure 32.19

```
#include <CGAL/basic.h>
#include <iostream>

#if (!CGAL_USE_CORE) && (!CGAL_USE_LEDA) && (!(CGAL_USE_GMP &&
    CGAL_USE_MPFI))
int main ()
{
  std::cout << "Sorry, this example needs CORE, LEDA, or GMP+MPFI ..."
            << std::endl;
  return 0;
}
#else

#include <CGAL/Arrangement_2.h>
#include <CGAL/Arr_algebraic_segment_traits_2.h>

#if CGAL_USE_GMP && CGAL_USE_MPFI
#include <CGAL/Gmpz.h>
typedef CGAL::Gmpz Integer;
#elif CGAL_USE_CORE
#include <CGAL/CORE_BigInt.h>
typedef CORE::BigInt Integer;
#else
#include <CGAL/leda_integer.h>
typedef LEDA::integer Integer;
```

```
#endif

typedef CGAL::Arr_algebraic_segment_traits_2<Integer> Arr_traits_2;
typedef CGAL::Arrangement_2<Arr_traits_2> Arrangement_2;
typedef Arr_traits_2::Curve_2 Curve_2;
typedef Arr_traits_2::Polynomial_2 Polynomial_2;

int main() {

    // For nice printouts
    CGAL::set_pretty_mode(std::cout);

    Arr_traits_2 arr_traits;

    // Functor to create a curve from a Polynomial_2
    Arr_traits_2::Construct_curve_2 construct_curve
        = arr_traits.construct_curve_2_object();

    Polynomial_2 x = CGAL::shift(Polynomial_2(1),1,0);
    Polynomial_2 y = CGAL::shift(Polynomial_2(1),1,1);

    Arrangement_2 arr(&arr_traits);

    // Construct an (unbounded line) with equation 3x-5y+2=0
    Polynomial_2 f1 = 3*x-5*y+2;
    Curve_2 cv1 = construct_curve(f1);
    std::cout << "Adding curve " << f1 << " to the arrangement" <<
   std::endl;
    CGAL::insert(arr,cv1);

    // Construct the ellipse x^2+3*y^2-10=0
    Polynomial_2 f2 = CGAL::ipower(x,2)+3*CGAL::ipower(y,2)-10;
    Curve_2 cv2 = construct_curve(f2);
    std::cout << "Adding curve " << f2 << " to the arrangement" <<
   std::endl;
    CGAL::insert(arr,cv2);

    // Construct a cubic curve with isoated point, and vertical asymptote
    // x^2+y^2+xy^2
    Polynomial_2 f3 =
   CGAL::ipower(x,2)+CGAL::ipower(y,2)+x*CGAL::ipower(y,2);
    Curve_2 cv3 = construct_curve(f3);
    std::cout << "Adding curve " << f3 << " to the arrangement" <<
   std::endl;
    CGAL::insert(arr,cv3);

    // Construct a curve of degree 6 with equation x^6+y^6-x^3y^3-12
    Polynomial_2 f4 = CGAL::ipower(x,6)+CGAL::ipower(y,6)-
                      CGAL::ipower(x,3)*CGAL::ipower(y,3)-12;
    Curve_2 cv4 = construct_curve(f4);
    std::cout << "Adding curve " << f4 << " to the arrangement" <<
   std::endl;
    CGAL::insert(arr,cv4);
```

```
    // Print the arrangement size.
    std::cout << "The arrangement size:" << std::endl
              << "   V = " << arr.number_of_vertices()
              << ",  E = " << arr.number_of_edges()
              << ",  F = " << arr.number_of_faces() << std::endl;

    return 0;
}

#endif
```

File: examples/Arrangement_on_surface_2/algebraic_curves.cpp

We first give a precise definition of segments of algebraic curves. A point p on a curve $C_f \subset \mathbb{R}^2$ (with f its defining equation) is called *semi-regular*, if locally around p, C_f can be written as a function graph of some continuous function in x or in y (we also say that p is parameterizable in x or y, respectively). The only two cases of non-semi-regular points are isolated points, and self-intersections. A *segment* of a curve is a closed and continuous point set such that each interior point is semi-regular. It follows that a weakly x-monotone segment is either a completely vertical segment, or a segment whose interior points are all parameterizable in x.

The traits class allows to construct weakly x-monotone segments of a curve using the *Construct_x_monotone_segment_2* functor. The *X_monotone_curve_2* type of the traits class represents weakly x-monotone segments of a curve; however, segments may need to be further subdivided into several (sub-)segments, for technical reasons. Therefore, *Construct_x_monotone_segment_2* constructs a sequence of *X_monotone_curve_2* objects, whose union represents the weakly x-monotone segment that was queried. We call a segment *terminal* if it can be represented by the type *X_monotone_curve_2*.

advanced

The subdivision of segments is due to the internal representation of x-monotone segments, which is based on a vertical decomposition. We assume the defining polynomial f of the curve C to be *square-free*, that means, it contains no divisor g^2 of total degree greater than zero. We define a *(complex) critical point* $p \in \mathbb{C}^2$ by

$$f(p) = 0 = \frac{\partial f}{\partial y}(p).$$

An x-coordinate $\alpha \in \mathbb{R}$ is *critical* if either some critical point has x-coordinate α, or if the leading coefficient of f, considered as a polynomial in y, vanishes. In particular, vertical lines of and isolated point of C can only take place at critical x-coordinates. Between two consecutive critical x-coordinates, the curve decomposes into a finite number of x-monotone segments (the same is true on the left of the leftmost, and on the right of the rightmost critical x-coordinate). The type *X_monotone_curve_2* is only able to represent such segments (and sub-segments of them). See Figure 32.20 for an example of a decomposition into terminal segments. Formally, a terminal segment is a weakly x-monotone segment that is either vertical, or its x-range contains no critical point in its interior.

advanced

Coordinates of points are represented by the type *Algebraic_real_1*, which is defined in the traits class. This type is taken from a model of the *AlgebraicKernel_1* concept, which is also available by the type *Algebraic_kernel_1*. One can use this model to create algebraic numbers as roots of univariate polynomials, and process them, for instance, compare them, or approximate them to any precision. See the documentation of *AlgebraicKernel_1* for more information. One can construct an object of type *Point_2* by a triple (x_0,cv,i), which means that the i-th point (counted from below) in the fiber of cv at the x-coordinate x_0 is constructed. This is also how points are presented internally. In the example displayed in Figure 32.20, if x_1 denotes the x-coordinate of p, and *cv*

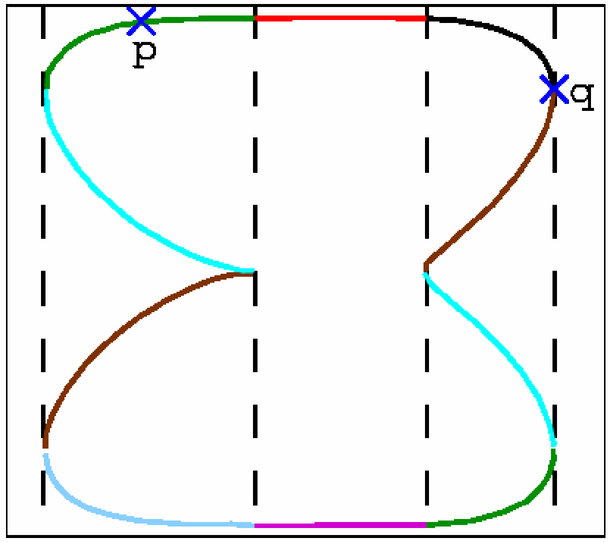


Figure 32.20: The critical x-coordinates of an algebraic curve (dashed lines), and its decomposition into terminal segments (in different colors). The segment from p to q consists of the union of three terminal segments.

represents the algebraic curve, then p could be represented by $(x_1, cv, 3)$. If x_2 is the x-coordinate of q, then $(x_2, cv, 1)$ is a valid representation of q. Although the y-coordinate of an object of type *Point_2* can be queried, we recommend to be careful with that option, since computing an explicit representation of the y-coordinate as an *Algebraic_real_1* object can become rather expensive.

The following code exemplifies various methods to construct algebraic segments. The computed arrangement is displayed in Figure 32.21.

```
#include <CGAL/basic.h>
#include <iostream>

#if (!CGAL_USE_CORE) && (!CGAL_USE_LEDA) && (!(CGAL_USE_GMP &&
    CGAL_USE_MPFI))
int main ()
{
  std::cout << "Sorry, this example needs CORE, LEDA, or GMP+MPFI ..."
            << std::endl;
  return 0;
}
#else

#include <CGAL/Arrangement_2.h>
#include <CGAL/Arr_algebraic_segment_traits_2.h>

#if CGAL_USE_GMP && CGAL_USE_MPFI
#include <CGAL/Gmpz.h>
typedef CGAL::Gmpz Integer;
#elif CGAL_USE_CORE
#include <CGAL/CORE_BigInt.h>
typedef CORE::BigInt Integer;
#else
#include <CGAL/leda_integer.h>
```

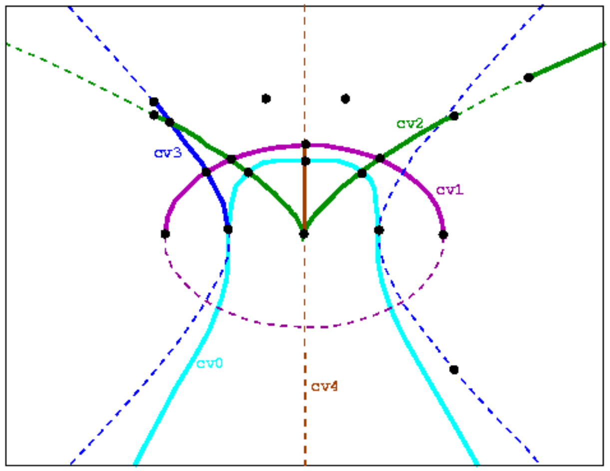


Figure 32.21: An arrangement of algebraic segments (solid lines), as constructed in *algebraic_segments.cpp*. The supporting curves are drawn in dashed lines.

```
typedef LEDA::integer Integer;
#endif

typedef CGAL::Arr_algebraic_segment_traits_2<Integer> Arr_traits_2;
typedef CGAL::Arrangement_2<Arr_traits_2> Arrangement_2;
typedef Arr_traits_2::Curve_2 Curve_2;
typedef Arr_traits_2::Polynomial_2 Polynomial_2;
typedef Arr_traits_2::Algebraic_real_1 Algebraic_real_1;
typedef Arr_traits_2::X_monotone_curve_2 X_monotone_curve_2;
typedef Arr_traits_2::Point_2 Point_2;


int main() {

    Arr_traits_2 arr_traits;

    Arr_traits_2::Construct_curve_2 construct_curve
        = arr_traits.construct_curve_2_object();
    Arr_traits_2::Construct_x_monotone_segment_2
   construct_x_monotone_segment
        = arr_traits.construct_x_monotone_segment_2_object();
    Arr_traits_2::Construct_point_2 construct_point
        = arr_traits.construct_point_2_object();
    Arr_traits_2::Make_x_monotone_2 make_x_monotone
        = arr_traits.make_x_monotone_2_object();

    Arrangement_2 arr(&arr_traits);

    std::vector<X_monotone_curve_2> segs;

    Polynomial_2 x = CGAL::shift(Polynomial_2(1),1,0);
    Polynomial_2 y = CGAL::shift(Polynomial_2(1),1,1);

    // Construct x^4+y^3-1
```

```
Curve_2 cv0 = construct_curve(CGAL::ipower(x,4)+CGAL::ipower(y,3)-1);
// Construct all x-monotone segments using the Make_x_mononotone
functor
std::vector<CGAL::Object> pre_segs;
make_x_monotone(cv0,std::back_inserter(pre_segs));
// Cast all CGAL::Objects into X_monotone_segment_2
// (the vector might also contain Point_2 objects for isolated
points,
// but not for this instance
for(size_t i = 0; i < pre_segs.size(); i++ ) {
     X_monotone_curve_2 curr;
     bool check = CGAL::assign(curr,pre_segs[i]);
     assert(check);
     segs.push_back(curr);
}
// Construct an ellipse with equation 2*x^2+5*y^2-7=0
Curve_2 cv1 =
construct_curve(2*CGAL::ipower(x,2)+5*CGAL::ipower(y,2)-7);

// Construct point on the upper arc (counting of arc numbers starts
with 0!
Point_2 p11 = construct_point(Algebraic_real_1(0),cv1,1);

construct_x_monotone_segment(cv1,p11,Arr_traits_2::POINT_IN_INTERIOR,
                             std::back_inserter(segs));

// Construct a vertical cusp x^2-y^3=0
Curve_2 cv2 = construct_curve(CGAL::ipower(x,2)-CGAL::ipower(y,3));

// Construct a segment containing the cusp point.
// This adds to X_monotone_curve_2 objects to the vector,
// because the cusp is a critical point
Point_2 p21 = construct_point(Algebraic_real_1(-2),cv2,0);
Point_2 p22 = construct_point(Algebraic_real_1(2),cv2,0);
construct_x_monotone_segment(cv2,p21,p22,std::back_inserter(segs));

// Construct an unbounded curve, starting at x=3
Point_2 p23 = construct_point(Algebraic_real_1(3),cv2,0);
construct_x_monotone_segment(cv2,p23,Arr_traits_2::MIN_ENDPOINT,
                             std::back_inserter(segs));

// Construct another conic: y^2-x^2+1
Curve_2 cv3 = construct_curve(CGAL::ipower(y,2)-CGAL::ipower(x,2)+1);

Point_2 p31 = construct_point(Algebraic_real_1(2),cv3,1);
construct_x_monotone_segment(cv3,p31,Arr_traits_2::MAX_ENDPOINT,
                             std::back_inserter(segs));

// Construct a vertical segment
Point_2 v1 = construct_point(0,0);
Point_2 v2 = construct_point(Algebraic_real_1(0),cv1,1);
construct_x_monotone_segment(v1,v2,std::back_inserter(segs));

CGAL::insert(arr,segs.begin(),segs.end());
```

```
  // Add some isolated points (must be wrapped into CGAL::Object)
  std::vector<CGAL::Object> isolated_points;
  isolated_points.push_back
      (CGAL::make_object(construct_point(Algebraic_real_1(2),cv3,0)));
  isolated_points.push_back
    (CGAL::make_object(construct_point(Integer(1),Integer(5))));
  isolated_points.push_back
    (CGAL::make_object(construct_point(Algebraic_real_1(-1),
                                        Algebraic_real_1(5))));

  CGAL::insert(arr,isolated_points.begin(), isolated_points.end());

  // Print the arrangement size.
  std::cout << "The arrangement size:" << std::endl
            << "   V = " << arr.number_of_vertices()
            << ",  E = " << arr.number_of_edges()
            << ",  F = " << arr.number_of_faces() << std::endl;

  return 0;
}

#endif
```

File: examples/Arrangement_on_surface_2/algebraic_segments.cpp

32.6.9 Traits-Class Decorators

Geometric traits-class decorators allow you to attach auxiliary data to curves and to points. The data is automatically manipulated by the decorators and distributed to the constructed geometric entities. Note that additional information can alternatively be maintained by extending the vertex, halfedge, or face types provided by the DCEL class used by the arrangement; see the details in Section 32.8.

The arrangement package includes a generic traits-class decorator template named *Arr_curve_data_traits_2<BaseTraits, XMonotoneCurveData, Merge, CurveData, Convert>*. This decorator is used to attach a data field to curves and to x-monotone curves. It is parameterized by a base-traits class, which is one of the geometric traits classes described in the previous subsections, or a user-defined traits class. The curve-data decorator derives itself from the base-traits class, and in particular inherits its *Point_2* type. In addition:

- *Curve_2* is derived from the basic *BaseTraits::Curve_2* class, extending it by an extra field of type *CurveData*.
- *X_monotone_curve_2* is derived from the basic *BaseTraits::X_monotone_curve_2* class, extending it by an extra field of type *XMonotoneCurveData*.

Note that the *Curve_2* and *X_monotone_curve_2* are not the same, even if the *BaseTraits::Curve_2* and *BaseTraits::X_monotone_curve_2* are (as in the case of the segment-traits class for example). The extended curve types support the additional methods *data()* and *set_data()* for accessing and modifying the data field.

You can create an extended curve (or an extended x-monotone curve) from a basic curve and a curve-data object. When curves are inserted into an arrangement, they may be split, and the decorator handles their data fields automatically:

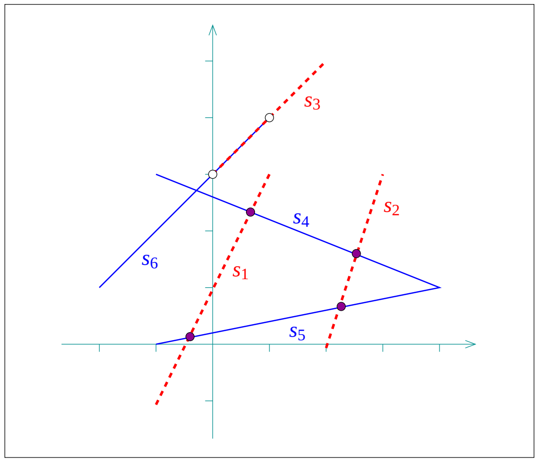


Figure 32.22: An arrangement of six red and blue segments, as constructed in *consolidated_curve_data.cpp*. Disks correspond to red–blue intersection points, while circles mark the endpoints of red–blue overlaps.

- When a curve is subdivided into x-monotone subcurves, its data field of type *CurveData* is converted to an *XMonotoneCurveData* object d using the *Convert* functor. The object d is automatically associated with each of the resulting x-monotone subcurves.

  Note that by default, the *CurveData* type is identical to the *XMonotoneCurveData* type (and the conversion functor *Convert* is trivially defined). Thus, the data field associated with the original curve is just duplicated and stored with the x-monotone subcurves.

- When an x-monotone curve is split into two, the decorator class automatically copies its data field to both resulting subcurves.

- When intersecting two x-monotone curves c_1 and c_2, the result may include overlapping sections, represented as x-monotone curves. In this case the data fields of c_1 and c_2 are merged into a single *XMonotoneCurveData* object, using the *Merge* functor, which is supplied as a parameter to the traits class-template. The resulting object is assigned to the data field of the overlapping subcurves.

- Merging two x-monotone curves is allowed only when (i) the two curves are geometrically mergeable — that is, the base-traits class allows to merge them — and (ii) the two curves store the same data field.

The *Arr_consolidated_curve_data_traits_2<BaseTraits, Data>* decorator specializes the generic curve-data decorator. It extends the basic *BaseTraits::Curve_2* by a single *Data* field, and the basic *BaseTraits::X_monotone_curve_2* with a *set* of (distinct) data objects. The *Data* type is required to support the equality operator, used to ensure that each set contains only distinct data objects with no duplicates. When a curve with a data field d is subdivided into x-monotone subcurves, each subcurve is associated with a set $S = \{d\}$. In case of an overlap between two x-monotone curves c_1 and c_2 with associated data sets S_1 and S_2, respectively, the overlapping subcurve is associated with the consolidated set $S_1 \cup S_2$.

Examples

In the following example, we use *Arr_segment_traits_2* as our base-traits class, attaching an additional *color* field to the segments using the consolidated curve-data traits class. A color may be either *blue* or *red*. Having constructed the arrangement of colored segments, as depicted in Figure 32.22, we detect the vertices that have incident edges mapped to both blue and red segments. Thus, they correspond to red–blue intersection points. We also locate the edge that corresponds to overlaps between red and blue line segments:

```
#include "arr_rational_nt.h"
#include <CGAL/Cartesian.h>
#include <CGAL/Arr_segment_traits_2.h>
#include <CGAL/Arr_consolidated_curve_data_traits_2.h>
#include <CGAL/Arrangement_2.h>
#include <CGAL/Arr_landmarks_point_location.h>

enum Segment_color {
  RED,
  BLUE
};

typedef CGAL::Cartesian<Number_type>                       Kernel;
typedef CGAL::Arr_segment_traits_2<Kernel>
    Segment_traits_2;
typedef Segment_traits_2::Curve_2                          Segment_2;
typedef CGAL::Arr_consolidated_curve_data_traits_2
                   <Segment_traits_2, Segment_color>       Traits_2;
typedef Traits_2::Point_2                                  Point_2;
typedef Traits_2::Curve_2
    Colored_segment_2;
typedef CGAL::Arrangement_2<Traits_2>                      Arrangement_2;
typedef CGAL::Arr_landmarks_point_location<Arrangement_2> Landmarks_pl;

int main ()
{
  // Construct an arrangement containing three RED line segments.
  Arrangement_2     arr;
  Landmarks_pl      pl (arr);

  Segment_2         s1 (Point_2(-1, -1), Point_2(1, 3));
  Segment_2         s2 (Point_2(2, 0), Point_2(3, 3));
  Segment_2         s3 (Point_2(0, 3), Point_2(2, 5));

  insert (arr, Colored_segment_2 (s1, RED), pl);
  insert (arr, Colored_segment_2 (s2, RED), pl);
  insert (arr, Colored_segment_2 (s3, RED), pl);

  // Insert three BLUE line segments.
  Segment_2         s4 (Point_2(-1, 3), Point_2(4, 1));
  Segment_2         s5 (Point_2(-1, 0), Point_2(4, 1));
  Segment_2         s6 (Point_2(-2, 1), Point_2(1, 4));

  insert (arr, Colored_segment_2 (s4, BLUE), pl);
  insert (arr, Colored_segment_2 (s5, BLUE), pl);
  insert (arr, Colored_segment_2 (s6, BLUE), pl);

  // Go over all vertices and print just the ones corresponding to
   intersection
  // points between RED segments and BLUE segments. Note that we skip
   endpoints
  // of overlapping sections.
  Arrangement_2::Vertex_const_iterator   vit;
```

```
Segment_color                          color;

for (vit = arr.vertices_begin(); vit != arr.vertices_end(); ++vit) {
  // Go over the incident halfedges of the current vertex and examine
 their
  // colors.
  bool        has_red = false;
  bool        has_blue = false;

  Arrangement_2::Halfedge_around_vertex_const_circulator  eit, first;

  eit = first = vit->incident_halfedges();
  do {
    // Get the color of the current half-edge.
    if (eit->curve().data().size() == 1) {
      color = eit->curve().data().front();

      if (color == RED)
        has_red = true;
      else if (color == BLUE)
        has_blue = true;
    }

    ++eit;
  } while (eit != first);

  // Print the vertex only if incident RED and BLUE edges were found.
  if (has_red && has_blue)
  {
    std::cout << "Red-blue intersection at (" << vit->point() << ")"
              << std::endl;
  }
}

// Locate the edges that correspond to a red-blue overlap.
Arrangement_2::Edge_iterator   eit;

for (eit = arr.edges_begin(); eit != arr.edges_end(); ++eit)
{
  // Go over the incident edges of the current vertex and examine their
  // colors.
  bool        has_red = false;
  bool        has_blue = false;

  Traits_2::Data_container::const_iterator       dit;

  for (dit = eit->curve().data().begin(); dit !=
 eit->curve().data().end();
       ++dit)
  {
    if (*dit == RED)
      has_red = true;
    else if (*dit == BLUE)
      has_blue = true;
```

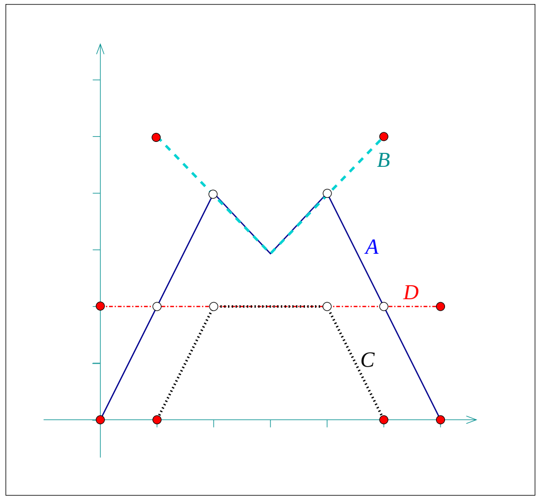


Figure 32.23: An arrangement of four polylines, named A–D, as constructed in *generic_curve_data.cpp*.

```
    }

    // Print the edge only if it corresponds to a red-blue overlap.
    if (has_red && has_blue)
      std::cout << "Red-blue overlap at [" << eit->curve() << "]"  <<
   std::endl;
  }
  return 0;
}
```

File: examples/Arrangement_on_surface_2/consolidated_curve_data.cpp

In the following example, we use *Arr_polyline_traits_2* as our base-traits class, attaching an additional *name* field to each polyline using the generic curve-data traits class. In case of overlaps, we simply concatenate the names of the overlapping polylines. Also notice how we replace the curve associated with the edges that correspond to overlapping polylines with geometrically equivalent curves, but with a different data fields:

```
#include "arr_rational_nt.h"
#include <CGAL/Cartesian.h>
#include <CGAL/Arr_segment_traits_2.h>
#include <CGAL/Arr_polyline_traits_2.h>
#include <CGAL/Arr_curve_data_traits_2.h>
#include <CGAL/Arrangement_2.h>
#include <string>

// Define a functor for concatenating name fields.
typedef std::string   Name;

struct Merge_names
{
  Name operator() (const Name& s1, const Name& s2) const
  {
```

```
    return (s1 + " " + s2);
  }
};

typedef CGAL::Cartesian<Number_type>                      Kernel;
typedef CGAL::Arr_segment_traits_2<Kernel>                Segment_traits_2;
typedef CGAL::Arr_polyline_traits_2<Segment_traits_2>
   Polyline_traits_2;
typedef Polyline_traits_2::Curve_2                        Polyline_2;
typedef CGAL::Arr_curve_data_traits_2<Polyline_traits_2, Name,
   Merge_names>
                                                          Traits_2;
typedef Traits_2::Point_2                                 Point_2;
typedef Traits_2::Curve_2                                 Curve_2;
typedef Traits_2::X_monotone_curve_2
   X_monotone_curve_2;
typedef CGAL::Arrangement_2<Traits_2>                     Arrangement_2;

int main ()
{
  // Construct an arrangement of four polylines named A--D.
  Arrangement_2    arr;

  Point_2          points1[5] = {Point_2(0,0), Point_2(2,4),
   Point_2(3,3),
                                 Point_2(4,4), Point_2(6,0)};
  insert (arr, Curve_2 (Polyline_2 (points1, points1 + 5), "A"));

  Point_2          points2[3] = {Point_2(1,5), Point_2(3,3),
   Point_2(5,5)};
  insert (arr, Curve_2 (Polyline_2 (points2, points2 + 3), "B"));

  Point_2          points3[4] = {Point_2(1,0), Point_2(2,2),
                                 Point_2(4,2), Point_2(5,0)};
  insert (arr, Curve_2 (Polyline_2 (points3, points3 + 4), "C"));

  Point_2          points4[2] = {Point_2(0,2), Point_2(6,2)};
  insert (arr, Curve_2 (Polyline_2 (points4, points4 + 2), "D"));

  // Print all edges that correspond to an overlapping polyline.
  Arrangement_2::Edge_iterator    eit;

  for (eit = arr.edges_begin(); eit != arr.edges_end(); ++eit) {
    if (eit->curve().data().length() > 1) {
      std::cout << "[" << eit->curve() << "]  "
                << "named: " << eit->curve().data() << std::endl;

      // Rename the curve associated with the edge.
      arr.modify_edge (eit, X_monotone_curve_2 (eit->curve(),
   "overlap"));
    }
  }
  return 0;
}
```

File: examples/Arrangement_on_surface_2/generic_curve_data.cpp

The third example we give in this section is based on *dual_lines.cpp* given in Section 32.5.2. It constructs the arrangement of the dual lines for a set of point given in an input file (by default we use *coll_points.dat*, which contains 50 points randomly selected on the grid $[-100,100] \times [-100,100]$; the file contains two distinct triplets of collinear points). Here we use the generic curve-data decorator to attach the index of the primal point to each of the lines. Doing so, we can go over the incident edges of each vertex whose degree is greater than 4 and report the subsets collinear points (if we have a vertex of degree d, we actually need to go over $\frac{d}{2}$ edges, as each incident line contributes exactly 2 edges). Note that in this case the dual line cannot overlap, so we use a dummy merge functor to instantiate the curve-data traits:

```
#include "arr_rational_nt.h"
#include <CGAL/Cartesian.h>
#include <CGAL/Arr_linear_traits_2.h>
#include <CGAL/Arr_curve_data_traits_2.h>
#include <CGAL/Arrangement_2.h>

typedef CGAL::Cartesian<Number_type>                         Kernel;
typedef CGAL::Arr_linear_traits_2<Kernel>                    Linear_traits_2;
typedef Linear_traits_2::Point_2                             Point_2;
typedef Linear_traits_2::Line_2                              Line_2;
typedef CGAL::Arr_curve_data_traits_2<Linear_traits_2,
                                      unsigned int>      Traits_2;
typedef Traits_2::X_monotone_curve_2
   X_monotone_curve_2;
typedef CGAL::Arrangement_2<Traits_2>                        Arrangement_2;

int main (int argc, char *argv[])
{
  // Get the name of the input file from the command line, or use the
   default
  // points.dat file if no command-line parameters are given.
  const char * filename = (argc > 1) ? argv[1] : "coll_points.dat";

  // Open the input file.
  std::ifstream     in_file (filename);

  if (! in_file.is_open()) {
    std::cerr << "Failed to open " << filename << " ..." << std::endl;
    return (1);
  }

  // Read the points from the file, and consturct their dual lines.
  std::vector<Point_2>            points;
  std::list<X_monotone_curve_2>   dual_lines;

  unsigned int n;
  in_file >> n;
  points.resize (n);
  unsigned int k;
  for (k = 0; k < n; ++k) {
    int px, py;
    in_file >> px >> py;
```

```
    points[k] = Point_2 (px, py);

    // The line dual to the point (p_x, p_y) is y = p_x*x - p_y,
    // or: p_x*x - y - p_y = 0:
    Line_2 dual_line = Line_2(Number_type(px),
                              Number_type(-1),
                              Number_type(-py));

    // Generate the x-monotone curve based on the line and the point
   index.
    dual_lines.push_back (X_monotone_curve_2 (dual_line, k));
  }
  in_file.close();

  // Construct the dual arrangement by aggragately inserting the lines.
  Arrangement_2      arr;

  insert (arr, dual_lines.begin(), dual_lines.end());

  // Look for vertices whose degree is greater than 4.
  Arrangement_2::Vertex_const_iterator                    vit;
  Arrangement_2::Halfedge_around_vertex_const_circulator  circ;
  unsigned int                                            d;

  for (vit = arr.vertices_begin(); vit != arr.vertices_end(); ++vit) {
    if (vit->degree() > 4) {
      // There should be vit->degree()/2 lines intersecting at the
   current
      // vertex. We print their primal points and their indices.
      circ = vit->incident_halfedges();
      for (d = 0; d < vit->degree() / 2; d++) {
        k = circ->curve().data();      // The index of the primal point.
        std::cout << "Point no. " << k+1 << ": (" << points[k] << "), ";
        ++circ;
      }
      std::cout << "are collinear." << std::endl;
    }
  }
  return 0;
}
```

File: examples/Arrangement_on_surface_2/dual_with_data.cpp

32.7 The Notification Mechanism

For some applications it is essential to know exactly what happens inside a specific arrangement-instance. For example, when a new curve is inserted into an arrangement, it might be desired to keep track of the faces that are split due to this insertion operation. Other important examples are the point-location strategies that require auxiliary data-structures (see Section 32.3.1), which must be notified on various local changes in the arrangement, in order to keep their data structures up-to-date. The arrangement package offers a mechanism that uses *observers* (see [GHJV95]) that can be attached to an arrangement instance and receive notifications about the changes this arrangement goes through.

The *Arr_observer<Arrangement>* class-template is parameterized with an arrangement class. It stores a pointer to an arrangement object, and is capable of receiving notifications *just before* a structural change occurs in the arrangement and *immediately after* such a change takes place. *Arr_observer* serves as a base class for other observer classes and defines a set of virtual notification functions, with default empty implementations.

The set of functions can be divided into three categories, as follows:

1. Notifiers of changes that affect the entire topological structure of the arrangement. This category consists of two pairs that notify the observer of the following changes:
   - The arrangement is cleared.
   - The arrangement is assigned with the contents of another arrangement.
2. Pairs of notifiers of a local change that occurs in the topological structure. Most notifier functions belong to this category. The relevant local changes include:
   - A new vertex is constructed and associated with a point.
   - An edge[19] is constructed and associated with an x-monotone curve.
   - An edge is split into two edges.
   - An existing face is split into two faces, as a consequence of the insertion of a new edge.
   - A hole is created in the interior of a face.
   - Two holes are merged to form a single hole, as a consequence of the insertion of a new edge.
   - A hole is moved from one face to another, as a consequence of a face split.
   - Two edges are merged into one edge.
   - Two faces are merged into one face, as a consequence of the removal of an edge that used to separate them.
   - One hole is split into two, as a consequence of the deletion of an edge that used to connect the two components.
   - A vertex is removed.
   - An edge is removed.
   - A hole is deleted from the interior of a face.
3. Notifiers about a change applied by a free (global) function. This category consists of a single pair of notifiers, namely *before_global_change()* and *after_global_change()*. Neither of these functions is invoked by methods of the *Arrangement_2* class. Instead, they are called by the free functions themselves. It is implied that no point-location queries (or any other queries for that matter) are issued between the calls to the notification functions above.

See the Reference Manual for a detailed specification of the *Arr_observer* class along with the exact prototypes of all notification functions.

Each arrangement object stores a (possibly empty) list of pointers to *Arr_observer* objects, and whenever one of the structural changes listed in the first two categories above is about to take place, the arrangement object performs a *forward* traversal on this list and invokes the appropriate function of each observer. After the change takes place the observer list is traversed in a *backward* manner (from tail to head), and the appropriate notification function is invoked for each observer. This allows the nesting of observer objects.

[19]The term “edge” refers here to a pair of twin half-edges.

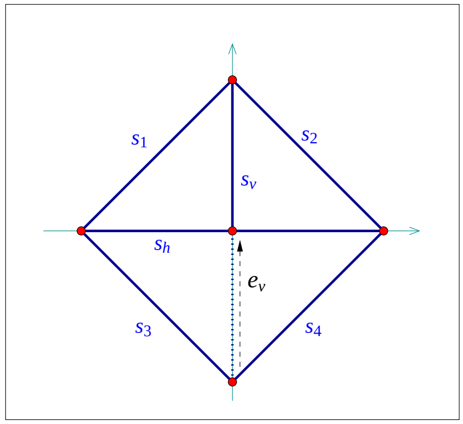


Figure 32.24: An arrangement of five line segments, as constructed in *observer.cpp*. The halfedge e_v (dashed) is eventually removed, so that the final arrangement consists of four faces (one unbounded and three bounded ones).

Concrete arrangement-observer classes should inherit from *Arr_observer*. When an observer is constructed, it is attached to a valid arrangement supplied to the observed constructor, or alternatively the observer can be attached to the arrangement at a later time. When this happens, the observer instance inserts itself into the observer list of the associated arrangement and starts receiving notifications whenever this arrangement changes thereafter. Naturally, the observer object unregisters itself by removing itself from this list just before it is destroyed.

The trapezoidal RIC and the landmark point-location strategies both use observers to keep their auxiliary data structures up-to-date. Besides them, users can define their own observer classes, by inheriting from the base observer class and overriding the relevant notification functions, as required by their applications.

The following example shows how to define and use an observer class. The observer in the example keeps track of the arrangement faces, and prints a message whenever a face is split into two due to the insertion of an edge, and whenever two faces merge into one due to the removal of an edge. The layout of the arrangement is depicted in Figure 32.24:

```
#include <CGAL/Cartesian.h>
#include <CGAL/Quotient.h>
#include <CGAL/MP_Float.h>
#include <CGAL/Arr_segment_traits_2.h>
#include <CGAL/Arrangement_2.h>
#include <CGAL/Arr_observer.h>

typedef CGAL::Quotient<CGAL::MP_Float>                    Number_type;
typedef CGAL::Cartesian<Number_type>                      Kernel;
typedef CGAL::Arr_segment_traits_2<Kernel>                Traits_2;
typedef Traits_2::Point_2                                 Point_2;
typedef Traits_2::X_monotone_curve_2                      Segment_2;
typedef CGAL::Arrangement_2<Traits_2>                     Arrangement_2;

// An arrangement observer, used to receive notifications of face splits
   and
// face mergers.
class My_observer : public CGAL::Arr_observer<Arrangement_2>
{
```

```
public:

  My_observer (Arrangement_2& arr) :
    CGAL::Arr_observer<Arrangement_2> (arr)
  {}

  virtual void before_split_face (Face_handle,
                                  Halfedge_handle e)
  {
    std::cout << "-> The insertion of :  [ " << e->curve()
              << " ]  causes a face to split." << std::endl;
  }

  virtual void before_merge_face (Face_handle,
                                  Face_handle,
                                  Halfedge_handle e)
  {
    std::cout << "-> The removal of :  [ " << e->curve()
              << " ]  causes two faces to merge." << std::endl;
  }

};

int main ()
{
  // Construct the arrangement containing one diamond-shaped face.
  Arrangement_2   arr;
  My_observer     obs (arr);

  Segment_2       s1 (Point_2(-1, 0), Point_2(0, 1));
  Segment_2       s2 (Point_2(0, 1), Point_2(1, 0));
  Segment_2       s3 (Point_2(1, 0), Point_2(0, -1));
  Segment_2       s4 (Point_2(0, -1), Point_2(-1, 0));

  insert_non_intersecting_curve (arr, s1);
  insert_non_intersecting_curve (arr, s2);
  insert_non_intersecting_curve (arr, s3);
  insert_non_intersecting_curve (arr, s4);

  // Insert a vertical segment dividing the diamond into two, and a
  // a horizontal segment further dividing the diamond into four:
  Segment_2       s_vert (Point_2(0, -1), Point_2(0, 1));
  Arrangement_2::Halfedge_handle
                  e_vert = insert_non_intersecting_curve (arr, s_vert);

  Segment_2       s_horiz (Point_2(-1, 0), Point_2(1, 0));

  insert (arr, s_horiz);

  std::cout << "The initial arrangement size:" << std::endl
            << "   V = " << arr.number_of_vertices()
            << ",  E = " << arr.number_of_edges()
            << ",  F = " << arr.number_of_faces() << std::endl;
```

```
  // Now remove a portion of the vertical segment.
  remove_edge (arr, e_vert);

  std::cout << "The final arrangement size:" << std::endl
            << "   V = " << arr.number_of_vertices()
            << ",  E = " << arr.number_of_edges()
            << ",  F = " << arr.number_of_faces() << std::endl;

  return 0;
}
```

File: examples/Arrangement_on_surface_2/observer.cpp

Observers are especially useful when the DCEL records are extended and store additional data, as they help updating this data on-line. See Section 32.8 for more details and examples.

32.8 Extending the DCEL

For many applications of the arrangement package it is necessary to store additional information (perhaps of non-geometric nature) with the arrangement cells. As vertices are associated with *Point_2* objects and edges (halfedge pairs) are associated with *X_monotone_curve_2* objects, both defined by the traits class, it is possible to extend the traits-class type by using a traits-class decorator, as explained in Section 32.6.9, which may be a sufficient solution for some applications. However, the DCEL faces are not associated with any geometric object, so it is impossible to extend them using a traits-class decorator. Extending the DCEL face records comes handy is such cases. As a matter of fact, it is possible to conveniently extend all DCEL records (namely vertices, halfedges and faces), which can also be advantageous for some applications.

All examples presented so far use the default *Arr_default_dcel<Traits>*. This is done implicitly, as this class serves as a default parameter for the *Arrangement_2* template. The default DCEL class just associates points with vertices and *x*-monotone curves with halfedge, but nothing more. In this section we show how to use alternative DCEL types to extend the desired DCEL records.

32.8.1 Extending the DCEL Faces

The *Arr_face_extended_dcel<Traits, FaceData>* class-template is used to associate auxiliary data field of type *FaceData* to each face record in the DCEL.

When an *Arrangement_2* object is parameterized by this DCEL class, its nested *Face* type is extended with the access function *data()* and with the modifier *set_data()*. Using these extra functions it is straightforward to access and maintain the auxiliary face-data field.

Note that the extra data fields must be maintained by the application programmers. They may choose to construct their arrangement, and only then go over the faces and attach the appropriate data fields to the arrangement faces. However, in some cases the face data can only be computed when the face is created (split from another face, or merged with another face). In such cases one can use an arrangement observer tailored for this task, which receives updates whenever a face is modified and sets its data field accordingly.

The next example constructs an arrangement that contains seven bounded faces induced by six line segments (see Figure 32.25). An observer gets notified each time a new face f is created and it associates f with a

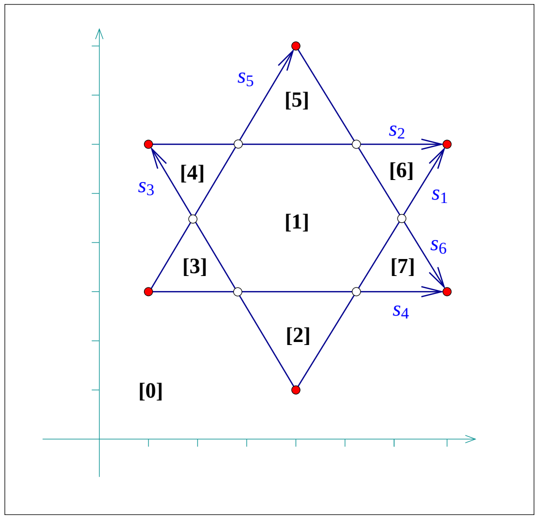


Figure 32.25: An arrangement of six line segments, as constructed in *face_extension.cpp* and *dcel_extension.cpp* (in *dcel_extension.cpp* we treat the segments as directed, so they are drawn as arrows directed from the source to the target). The indices associated with the halfedges in *face_extension.cpp* are shown in brackets.

running index, (where the index of the unbounded face is 0). As a result, the faces are numbered according to their creation order, as one can easily verify by examining the insertion order of the segments:[20]

```
#include "arr_rational_nt.h"
#include <CGAL/Cartesian.h>
#include <CGAL/Arr_segment_traits_2.h>
#include <CGAL/Arrangement_2.h>
#include <CGAL/Arr_extended_dcel.h>
#include <CGAL/Arr_observer.h>

typedef CGAL::Cartesian<Number_type>                          Kernel;
typedef CGAL::Arr_segment_traits_2<Kernel>                    Traits_2;
typedef Traits_2::Point_2                                     Point_2;
typedef Traits_2::X_monotone_curve_2                          Segment_2;
typedef CGAL::Arr_face_extended_dcel<Traits_2, int>           Dcel;
typedef CGAL::Arrangement_2<Traits_2, Dcel>                   Arrangement_2;

// An arrangement observer, used to receive notifications of face splits
   and
// to update the indices of the newly created faces.
class Face_index_observer : public CGAL::Arr_observer<Arrangement_2>
{
private:
  int      n_faces;          // The current number of faces.

public:

  Face_index_observer (Arrangement_2& arr) :
    CGAL::Arr_observer<Arrangement_2> (arr),
```

[20]For simplicity, the particular observer used must be attached to an empty arrangement. It is not difficult however to modify the program to handle the general case of attaching a similar observer to a non-empty arrangement.

```
    n_faces (0)
  {
    CGAL_precondition (arr.is_empty());

    arr.unbounded_face()->set_data (0);
    n_faces++;
  }

  virtual void after_split_face (Face_handle /* old_face */,
                                 Face_handle new_face, bool )
  {
    // Assign index to the new face.
    new_face->set_data (n_faces);
    n_faces++;
  }
};

int main ()
{
  // Construct the arrangement containing two intersecting triangles.
  Arrangement_2          arr;
  Face_index_observer    obs (arr);

  Segment_2      s1 (Point_2(4, 1), Point_2(7, 6));
  Segment_2      s2 (Point_2(1, 6), Point_2(7, 6));
  Segment_2      s3 (Point_2(4, 1), Point_2(1, 6));
  Segment_2      s4 (Point_2(1, 3), Point_2(7, 3));
  Segment_2      s5 (Point_2(1, 3), Point_2(4, 8));
  Segment_2      s6 (Point_2(4, 8), Point_2(7, 3));

  insert_non_intersecting_curve (arr, s1);
  insert_non_intersecting_curve (arr, s2);
  insert_non_intersecting_curve (arr, s3);
  insert (arr, s4);
  insert (arr, s5);
  insert (arr, s6);

  // Go over all arrangement faces and print the index of each face and
   it
  // outer boundary. The face index is stored in its data field in our
   case.
  Arrangement_2::Face_const_iterator            fit;
  Arrangement_2::Ccb_halfedge_const_circulator  curr;

  std::cout << arr.number_of_faces() << " faces:" << std::endl;
  for (fit = arr.faces_begin(); fit != arr.faces_end(); ++fit) {
    std::cout << "Face no. " << fit->data() << ": ";
    if (fit->is_unbounded())
      std::cout << "Unbounded." << std::endl;
    else {
      curr = fit->outer_ccb();
      std::cout << curr->source()->point();
      do {
        std::cout << " --> " << curr->target()->point();
```

```
        ++curr;
      } while (curr != fit->outer_ccb());
      std::cout << std::endl;
    }
  }

  return 0;
}
```

File: examples/Arrangement_on_surface_2/face_extension.cpp

32.8.2 Extending All DCEL Records

The *Arr_extended_dcel<Traits, VertexData, HalfedgeData, FaceData>* class-template is used to associate auxiliary data fields of types *VertexData HalfedgeData*, and *FaceData* to each DCEL vertex, halfedge, and face record types, respectively.

When an *Arrangement_2* object is injected with this DCEL class, each one of its nested *Vertex*, *Halfedge* and *Face* classes is extended by the access function *data()* and by the modifier *set_data()*.

The next example shows how to use a DCEL with extended vertex, halfedge, and face records. In this example each vertex is associated with a color, which may be blue, red, or white, depending on whether the vertex is isolated, represents a segment endpoint, or whether it represents an intersection point. Each halfedge is associated with Boolean flag indicating whether its direction is the same as the direction of its associated segment (in this example segments are treated as directed objects). Each face is also extended to store the size of its outer boundary.

The constructed arrangement, depicted in Figure 32.25, is similar to the arrangement constructed in the previous example. Note that all auxiliary data fields are set during the construction phase. Also note that the data fields are properly maintained when the arrangement is copied to another arrangement instance:

```
#include "arr_rational_nt.h"
#include <CGAL/Cartesian.h>
#include <CGAL/Arr_segment_traits_2.h>
#include <CGAL/Arrangement_2.h>
#include <CGAL/Arr_extended_dcel.h>

enum Color {BLUE, RED, WHITE};

typedef CGAL::Cartesian<Number_type>                              Kernel;
typedef CGAL::Arr_segment_traits_2<Kernel>                        Traits_2;
typedef Traits_2::Point_2                                         Point_2;
typedef Traits_2::X_monotone_curve_2
   Segment_2;
typedef CGAL::Arr_extended_dcel<Traits_2,Color, bool, int>       Dcel;
typedef CGAL::Arrangement_2<Traits_2, Dcel>
   Arrangement_2;

int main ()
{
  // Construct the arrangement containing two intersecting triangles.
```

```
Arrangement_2          arr;

Segment_2      s1 (Point_2(4, 1), Point_2(7, 6));
Segment_2      s2 (Point_2(1, 6), Point_2(7, 6));
Segment_2      s3 (Point_2(4, 1), Point_2(1, 6));
Segment_2      s4 (Point_2(1, 3), Point_2(7, 3));
Segment_2      s5 (Point_2(1, 3), Point_2(4, 8));
Segment_2      s6 (Point_2(4, 8), Point_2(7, 3));

insert_non_intersecting_curve (arr, s1);
insert_non_intersecting_curve (arr, s2);
insert_non_intersecting_curve (arr, s3);
insert (arr, s4);
insert (arr, s5);
insert (arr, s6);

// Go over all arrangement vertices and set their colors according to
 our
// coloring convention.
Arrangement_2::Vertex_iterator            vit;
unsigned int                              degree;

for (vit = arr.vertices_begin(); vit != arr.vertices_end(); ++vit)
{
  degree = vit->degree();
  if (degree == 0)
    vit->set_data (BLUE);       // Isolated vertex.
  else if (degree <= 2)
    vit->set_data (RED);        // Vertex represents an endpoint.
  else
    vit->set_data (WHITE);      // Vertex represents an intersection
 point.
}

// Go over all arrangement edges and set their flags.
Arrangement_2::Edge_iterator              eit;
bool                                      flag;

for (eit = arr.edges_begin(); eit != arr.edges_end(); ++eit) {
  // Check if the halfegde has the same diretion as its associated
  // segment. Note that its twin always has an opposite direction.
  flag = (eit->source()->point() == eit->curve().source());
  eit->set_data (flag);
  eit->twin()->set_data (!flag);
}

// For each arrangement face, print the outer boundary and its size.
Arrangement_2::Face_iterator              fit;
Arrangement_2::Ccb_halfedge_circulator    curr;
int                                       boundary_size;

for (fit = arr.faces_begin(); fit != arr.faces_end(); ++fit) {
  boundary_size = 0;
  if (! fit->is_unbounded()) {
```

```
      curr = fit->outer_ccb();
      do {
        ++boundary_size;
        ++curr;
      } while (curr != fit->outer_ccb());
    }
    fit->set_data (boundary_size);
  }

  // Copy the arrangement and print the vertices.
  Arrangement_2    arr2 = arr;

  std::cout << "The arrangement vertices:" << std::endl;
  for (vit = arr2.vertices_begin(); vit != arr2.vertices_end(); ++vit) {
    std::cout << '(' << vit->point() << ") - ";
    switch (vit->data()) {
     case BLUE  : std::cout << "BLUE."  << std::endl; break;
     case RED   : std::cout << "RED."   << std::endl; break;
     case WHITE : std::cout << "WHITE." << std::endl; break;
    }
  }

  return 0;
}
```

File: examples/Arrangement_on_surface_2/dcel_extension.cpp

advanced

The various DCEL classes presented in this section are perfectly sufficient for most applications based on the arrangement package. However, users may also use their own implementation of a DCEL class to instantiate the *Arrangement_2* class-template, in case they need special functionality from their DCEL. Such a class must be a model of the concept *ArrangementDcel*, whose exact specification is listed in the Reference Manual.

advanced

32.9 Overlaying Arrangements

Assume that we are given two geographic maps represented as arrangements with some data objects attached to their faces, representing some geographic information — for example, a map of the annual precipitation in some country and a map of the vegetation in the same country. It is interesting to overlay the two maps to locate, for example, the regions where there is a pine forest and the annual precipitation is between 1000 mm and 1500 mm.

Computing the overlay of two planar arrangement is also useful for supporting Boolean set operations on polygons (or generalized polygons, see, e.g., [BEH+02]).

The function *overlay (arr_a, arr_b, ovl_arr, ovl_traits)* accepts two input arrangement instances *arr_a* and *arr_b*, and constructs their overlay instance *ovl_arr*. All three arrangements must use the same geometric primitives. More precisely, let *arr_a* be of type *Arrangement_2<Traits_A,Dcel_A>*, *arr_b* be of type *Arrangement_2<Traits_B,Dcel_B>* and the resulting *ovl_arr* be of type *Arrangement_2<Traits_R,Dcel_R>*. All types nested in geometry traits *Traits_A*, e.g., *Point_2* and *X_monotone_curve_2*, must be convertible to the corresponding types nested in geometry traits *Traits_R*. The same holds for all types nested in geometry traits *Traits_B*. The *ovl_traits* parameter is an instance of an *overlay traits-class*, which enables the creation of *Dcel_R* records in the overlaid arrangement from the DCEL features of *arr_a* and *arr_b* that they correspond to.

In principle, we distinguish between three levels of overlay:

Simple overlay: An overlay of two arrangements that store no additional data with their DCEL records. That is, they are defined using the default DCEL class *Arr_default_dcel*. Typically, the overlaid arrangement in this case stores no extra data with its DCEL records as well (or if it does, the additional data fields cannot be computed by the overlay operation), so by overlaying the two arrangement we just compute the arrangement of all curves that induce *arr_a* and *arr_b*. Note that the same result can be obtained using the standard insertion operations, but users may choose to use overlay computation in order to achieve better running times.

The *Arr_default_overlay_traits* class should be used as an overlay traits-class for such simple overlay operations.

Face overlay: An overlay of two arrangements that store additional data fields with their faces (e.g., the geographic-map example given in the beginning of this section). The resulting overlaid arrangement typically also stores extraneous data fields with its faces, where the data field that is attached to an overlaid face can be computed from the data fields of the two faces (in *arr_a* and *arr_b*) that induce the overlaid face.

The *Arr_face_overlay_traits* class should be used as an overlay traits-class for face-overlay operations. It operates on arrangement, whose DCEL representation is based on the *Arr_face_extended_dcel* class-template (see Section 32.8.1). The face-overlay traits-class is parameterized by a functor that is capable of combining two face-data fields of types *Dcel_A::Face_data* and *Dcel_B::Face_data*, and computing the output *Dcel_R::Face_data* object. The overlay traits-class uses this functor to properly construct the overlaid faces.

Full overlay: An overlay of two arrangements that store additional data fields with all their DCEL records. That is, their DCEL classes are instantiations of the *Arr_extended_dcel* class-template (see Section 32.8.2), where the resulting arrangement also extends it DCEL records with data fields computed on the basis of the overlapping DCEL features of the two input arrangements.

In the following subsections we give some examples for the simple and the face-overlay operations and demonstrate how to use the auxiliary overlay traits-classes. For the full overlay operations users need to implement their specialized overlay traits-class, which models the *OverlayTraits* concept. The details of this concept are given in the Reference Manual.

32.9.1 Example for a Simple Overlay

The next program constructs two simple arrangements, as depicted in Figure 32.26 and computes their overlay:

```
#include "arr_rational_nt.h"
#include <CGAL/Cartesian.h>
#include <CGAL/Arr_segment_traits_2.h>
#include <CGAL/Arrangement_2.h>
#include <CGAL/Arr_overlay_2.h>
#include <CGAL/Arr_default_overlay_traits.h>

typedef CGAL::Cartesian<Number_type>                    Kernel;
```

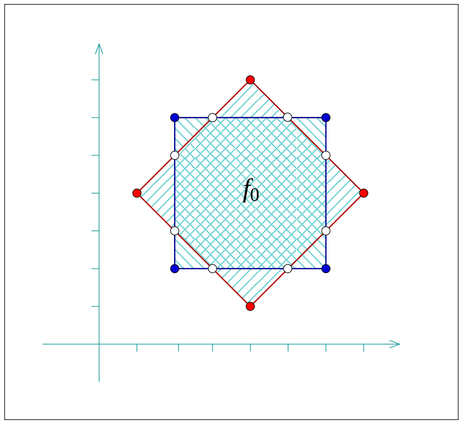


Figure 32.26: Overlaying two simple arrangements of line segments, as done in *overlay.cpp* and *ex_face_extension_overlay.cpp*. In *face_extension_overlay.cpp* the two bounded faces are considered as *marked*, and the octagonal face which is the intersection of the two marked faces is denoted by f_0.

```
typedef CGAL::Arr_segment_traits_2<Kernel>                    Traits_2;
typedef Traits_2::Point_2                                     Point_2;
typedef Traits_2::X_monotone_curve_2                          Segment_2;
typedef CGAL::Arrangement_2<Traits_2>                         Arrangement_2;
typedef CGAL::Arr_default_overlay_traits<Arrangement_2>  Overlay_traits;

int main ()
{
  // Construct the first arrangement, containing a square-shaped face.
  Arrangement_2          arr1;

  Segment_2      s1 (Point_2(2, 2), Point_2(6, 2));
  Segment_2      s2 (Point_2(6, 2), Point_2(6, 6));
  Segment_2      s3 (Point_2(6, 6), Point_2(2, 6));
  Segment_2      s4 (Point_2(2, 6), Point_2(2, 2));

  insert_non_intersecting_curve (arr1, s1);
  insert_non_intersecting_curve (arr1, s2);
  insert_non_intersecting_curve (arr1, s3);
  insert_non_intersecting_curve (arr1, s4);

  // Construct the second arrangement, containing a rhombus-shaped face.
  Arrangement_2          arr2;

  Segment_2      t1 (Point_2(4, 1), Point_2(7, 4));
  Segment_2      t2 (Point_2(7, 4), Point_2(4, 7));
  Segment_2      t3 (Point_2(4, 7), Point_2(1, 4));
  Segment_2      t4 (Point_2(1, 4), Point_2(4, 1));

  insert_non_intersecting_curve (arr2, t1);
  insert_non_intersecting_curve (arr2, t2);
  insert_non_intersecting_curve (arr2, t3);
  insert_non_intersecting_curve (arr2, t4);

  // Compute the overlay of the two arrangements.
```

```
  Arrangement_2          overlay_arr;
  Overlay_traits         overlay_traits;

  overlay (arr1, arr2, overlay_arr, overlay_traits);

  // Print the size of the overlaid arrangement.
  std::cout << "The overlaid arrangement size:" << std::endl
            << "   V = " << overlay_arr.number_of_vertices()
            << ",  E = " << overlay_arr.number_of_edges()
            << ",  F = " << overlay_arr.number_of_faces() << std::endl;

  return 0;
}
```

File: examples/Arrangement_on_surface_2/overlay.cpp

32.9.2 Examples for a Face Overlay

The following example shows how to compute the intersection of two polygons using the *overlay()* function. It uses a face-extended DCEL class to define our arrangement class. The DCEL extends each face with a Boolean flag. A polygon is represented as a *marked* arrangement face, (whose flag is set). The example uses a face-overlay traits class, instantiated with a functor that simply performs a logical *and* operations on Boolean flags. As a result, a face in the overlaid arrangement is marked only when it corresponds to an overlapping region of two marked cells in the input arrangements. Namely, it is part of the intersection of the two polygons.

The example computes the intersection between a square and a rhombus, (which is actually also a square). The resulting polygon is an octagon, denoted by f_0 in Figure 32.26:

```
#include "arr_rational_nt.h"
#include <CGAL/Cartesian.h>
#include <CGAL/Arr_segment_traits_2.h>
#include <CGAL/Arrangement_2.h>
#include <CGAL/Arr_extended_dcel.h>
#include <CGAL/Arr_overlay_2.h>
#include <CGAL/Arr_default_overlay_traits.h>

typedef CGAL::Cartesian<Number_type>                          Kernel;
typedef CGAL::Arr_segment_traits_2<Kernel>                    Traits_2;
typedef Traits_2::Point_2                                     Point_2;
typedef Traits_2::X_monotone_curve_2
   Segment_2;
typedef CGAL::Arr_face_extended_dcel<Traits_2, bool>          Dcel;
typedef CGAL::Arrangement_2<Traits_2, Dcel>
   Arrangement_2;
typedef CGAL::Arr_face_overlay_traits<Arrangement_2,
                                      Arrangement_2,
                                      Arrangement_2,
                                      std::logical_and<bool> >
   Overlay_traits;

int main ()
```

```
{
  // Construct the first arrangement, containing a square-shaped face.
  Arrangement_2          arr1;

  Segment_2      s1 (Point_2(2, 2), Point_2(6, 2));
  Segment_2      s2 (Point_2(6, 2), Point_2(6, 6));
  Segment_2      s3 (Point_2(6, 6), Point_2(2, 6));
  Segment_2      s4 (Point_2(2, 6), Point_2(2, 2));

  insert_non_intersecting_curve (arr1, s1);
  insert_non_intersecting_curve (arr1, s2);
  insert_non_intersecting_curve (arr1, s3);
  insert_non_intersecting_curve (arr1, s4);

  // Mark just the bounded face.
  Arrangement_2::Face_iterator   fit;

  CGAL_assertion (arr1.number_of_faces() == 2);
  for (fit = arr1.faces_begin(); fit != arr1.faces_end(); ++fit)
    fit->set_data (fit != arr1.unbounded_face());

  // Construct the second arrangement, containing a rhombus-shaped face.
  Arrangement_2          arr2;

  Segment_2      t1 (Point_2(4, 1), Point_2(7, 4));
  Segment_2      t2 (Point_2(7, 4), Point_2(4, 7));
  Segment_2      t3 (Point_2(4, 7), Point_2(1, 4));
  Segment_2      t4 (Point_2(1, 4), Point_2(4, 1));

  insert_non_intersecting_curve (arr2, t1);
  insert_non_intersecting_curve (arr2, t2);
  insert_non_intersecting_curve (arr2, t3);
  insert_non_intersecting_curve (arr2, t4);

  // Mark just the bounded face.
  CGAL_assertion (arr2.number_of_faces() == 2);
  for (fit = arr2.faces_begin(); fit != arr2.faces_end(); ++fit)
    fit->set_data (fit != arr2.unbounded_face());

  // Compute the overlay of the two arrangements, marking only the faces
   that
  // are intersections of two marked faces in arr1 and arr2,
   respectively.
  Arrangement_2          overlay_arr;
  Overlay_traits         overlay_traits;

  overlay (arr1, arr2, overlay_arr, overlay_traits);

  // Go over the faces of the overlaid arrangement and print just the
   marked
  // ones.
  Arrangement_2::Ccb_halfedge_circulator    curr;

  std::cout << "The union is: ";
```

```
  for (fit = overlay_arr.faces_begin(); fit != overlay_arr.faces_end();
   ++fit) {
    if (! fit->data())
      continue;

    curr = fit->outer_ccb();
    std::cout << curr->source()->point();
    do {
      std::cout << " --> " << curr->target()->point();
      ++curr;
    } while (curr != fit->outer_ccb());
    std::cout << std::endl;
  }

  return 0;
}
```

File: examples/Arrangement_on_surface_2/face_extension_overlay.cpp

The next example demonstrates the face overlay of two arrangements that have unbounded faces as well. The first arrangement is formed by the two lines $y = x$ and $y = -x$, that subdivide the plane into four unbounded faces, denoted A, B, C and D. The second arrangement comprises four line segments that form a square-shaped face. When we overlay the two arrangements, each of the four faces A, B, C and D is split into an unbounded face (indexed 1) and a bounded face (indexed 2):

```
#include <string>
#include <boost/lexical_cast.hpp>

#include <CGAL/Exact_predicates_exact_constructions_kernel.h>
#include <CGAL/Arr_linear_traits_2.h>
#include <CGAL/Arrangement_2.h>
#include <CGAL/Arr_extended_dcel.h>
#include <CGAL/Arr_overlay_2.h>
#include <CGAL/Arr_default_overlay_traits.h>

// Define a functor for creating a label from a characer and an integer.
struct Overlay_label
{
  std::string operator() (char c, int i) const
  {
    return boost::lexical_cast<std::string>(c) +
      boost::lexical_cast<std::string>(i);
  }
};

typedef CGAL::Exact_predicates_exact_constructions_kernel   Kernel;
typedef CGAL::Arr_linear_traits_2<Kernel>                   Traits_2;
typedef Traits_2::Point_2                                   Point_2;
typedef Traits_2::Segment_2                                 Segment_2;
typedef Traits_2::Ray_2                                     Ray_2;
typedef Traits_2::Line_2                                    Line_2;
```

```
typedef Traits_2::X_monotone_curve_2
   X_monotone_curves_2;
typedef CGAL::Arr_face_extended_dcel<Traits_2, char>        DcelA;
typedef CGAL::Arrangement_2<Traits_2, DcelA>
   ArrangementA_2;
typedef CGAL::Arr_face_extended_dcel<Traits_2, int>         DcelB;
typedef CGAL::Arrangement_2<Traits_2, DcelB>
   ArrangementB_2;
typedef CGAL::Arr_face_extended_dcel<Traits_2, std::string> DcelRes;
typedef CGAL::Arrangement_2<Traits_2, DcelRes>
   ArrangementRes_2;
typedef CGAL::Arr_face_overlay_traits<ArrangementA_2,
                                      ArrangementB_2,
                                      ArrangementRes_2,
                                      Overlay_label>
   Overlay_traits;

int main ()
{
  // Construct the first arrangement, induced by two line y = x and y =
   -x.
  ArrangementA_2          arr1;

  insert (arr1, Line_2 (Point_2(0, 0), Point_2(1, 1)));
  insert (arr1, Line_2 (Point_2(0, 0), Point_2(1, -1)));

  // Label the four (unbounded) face of the arrangement as 'A' to 'D'.
  // We do so by traversing the incident faces to the halfedges aroung
   the
  // single arrangement vertex (0, 0).
  CGAL_assertion (arr1.number_of_vertices() == 1);

  ArrangementA_2::Halfedge_around_vertex_circulator  first, curr;
  char                                               clabel = 'A';

  curr = first = arr1.vertices_begin()->incident_halfedges();
  do {
    curr->face()->set_data (clabel);
    ++clabel;
    ++curr;
  } while (curr != first);
  std::cout << "Done with arr1." << std::endl;

  // Construct the second arrangement, containing a single square-shaped
   face.
  ArrangementB_2          arr2;

  insert (arr2, Segment_2 (Point_2(-4, -4), Point_2(4, -4)));
  insert (arr2, Segment_2 (Point_2(4, -4), Point_2(4, 4)));
  insert (arr2, Segment_2 (Point_2(4, 4), Point_2(-4, 4)));
  insert (arr2, Segment_2 (Point_2(-4, 4), Point_2(-4, -4)));

  // Give the unbounded face the index 1, and the bounded face the index
   2.
```

```
  CGAL_assertion (arr2.number_of_faces() == 2);

  ArrangementB_2::Face_iterator    fit;

  for (fit = arr2.faces_begin(); fit != arr2.faces_end(); ++fit)
    fit->set_data ((fit == arr2.unbounded_face()) ? 1 : 2);
  std::cout << "Done with arr2." << std::endl;

  // Compute the overlay of the two arrangements.
  ArrangementRes_2       overlay_arr;
  Overlay_traits         overlay_traits;

  overlay (arr1, arr2, overlay_arr, overlay_traits);

  // Go over the faces of the overlaid arrangement and their labels.
  ArrangementRes_2::Face_iterator  res_fit;

  std::cout << "The overlay faces are: ";
  for (res_fit = overlay_arr.faces_begin();
       res_fit != overlay_arr.faces_end(); ++res_fit)
  {
    std::cout << res_fit->data() << " ("
              << (res_fit->is_unbounded() ? "unbounded" : "bounded")
              << ")." << std::endl;
  }

  return 0;
}
```

File: examples/Arrangement_on_surface_2/overlay_unbounded.cpp

32.10 Storing the Curve History

As stated at the beginning of this chapter (Section 32.1), when one constructs an arrangement induced by a set $\mathcal{C}$ of arbitrary planar curves, she or he constructs a collection $\mathcal{C}''$ of x-monotone subcurves of $\mathcal{C}$ that are pairwise disjoint in their interior, and these subcurves are associated with the arrangement edges (more precisely, with the DCEL halfedges). Doing so, the connection between the originating input curves and the arrangement edges is lost. This loss might be acceptable for some applications. However, in many practical cases it is important to determine the input curves that give rise to the final subcurves.

The *Arrangement_with_history_2<Traits,Dcel>* class-template extends the *Arrangement_2* class by keeping an additional container of input curves representing $\mathcal{C}$, and by maintaining a cross-mapping between these curves and the arrangement edges they induce. The traits class that is used for instantiating the template should be a model of the *ArrangementTraits_2* concept (see Section 32.4.1). That is, it should define the *Curve_2* type (and not just the *X_monotone_curve_2* type). The *Dcel* parameter should model the *ArrangementDcel* concept. Users can use the default DCEL class or an extended DCEL class according to their needs.

32.10.1 Traversing an Arrangement with History

The *Arrangement_with_history_2* class extends the *Arrangement_2* class, thus all the iterator and circulator types that are defined by the arrangement class are also available in *Arrangement_with_history_2*. The reader is referred to Section 32.2.2 for a comprehensive review of these functions.

As mentioned above, the *Arrangement_with_history_2* class maintains a container of input curves, which can be accessed using curve handles. The member function *number_of_curves()* returns the number of input curves stored in the container, while *curves_begin()* and *curves_end()* return *Arrangement_with_history_2::Curve_iterator* objects that define the valid range of curves that induce the arrangement. The value type of this iterator is *Curve_2*. Moreover, the curve-iterator type is equivalent to *Arrangement_with_history_2::Curve_handle*, which is used for accessing the stored curves. Conveniently, the corresponding constant-iterator and constant-handle types are also defined.

As mentioned in the previous paragraph, a *Curve_handle* object *ch* serves as a pointer to a curve stored in an arrangement-with-history instance *arr*. Using this handle, it is possible to obtain the number of arrangement edges this curve induces by calling *arr.number_of_induced_edges(ch)*. The functions *arr.induced_edges_begin(ch)* and *arr.induced_edges_end(ch)* return iterators of type *Arrangement_with_history_2::Induced_edges_iterator* that define the valid range of edges induced by *ch*. The value type of these iterators is *Halfedge_handle*. It is thus possible to traverse all arrangement edges induced by an input curve.

It is also important to be able to perform the inverse mapping. Given an arrangement edge, we would like to be able to determine which input curve induces it. In case the edge represents an overlap of several curves, we should be able to trace all input curves that overlap over this edge. The *Arrangement_with_history_2* class is extended by several member functions that enable such an inverse mapping. Given a halfedge handle *e* in an arrangement with history *arr*, then *arr.number_of_originating_curves(e)* returns the number of curves that induce the edge (which should be 1 in non-degenerate cases, and 2 or more in case of overlaps), while *arr.originating_curves_begin(e)* and *arr.originating_curves_end(e)* return *Arrangement_with_history_2::Originating_curve_iterator* objects that define the range of curves that induce *e*. The value type of these iterator is *Curve_2*.

It is possible to overlay two *Arrangement_with_history_2* instances instantiated by the same traits class. In this case, the resulting arrangement will store a consolidated container of input curves, and automatically preserve the cross-mapping between the arrangement edges and the consolidated curve set. Users can employ an overlay-traits class to maintain any type of auxiliary data stored with the DCEL features (see Section 32.9).

32.10.2 Modifying an Arrangement with History

As the *Arrangement_with_history_2* class extends the *Arrangement_2* class, it inherits the fundamental modification operations, such as *assign()* and *clear()*, from it. The vertex-manipulation functions are also inherited and supported (see Sections 32.2.3 and 32.4.1 for the details). However, there are some fundamental differences between the interfaces of the two classes, which we highlight in this subsection.

The most significant difference between the arrangement-with-history class and the basic arrangement class is the way they handle their input curves. *Arrangement_with_history_2* always stores the *Curve_2* objects that induce it, thus it is impossible to insert x-monotone curves into an arrangement with history. The free *insert_non_intersecting_curve()* and *insert()* that receives x-monotone curve (as well as their aggregated versions) are therefore not available for arrangement-with-history instances and only the free *insert()* and *insert()* functions that receive *Curve_2* (the incremental insertion function and the aggregated insertion function) are supported — see also Section 32.4.1. Notice however that while the incremental insertion function *insert(arr,c)* for an *Arrangement_2* object *arr* does not have a return value, the corresponding arrangement-with-history function returns a *Curve_handle* to the inserted curve.

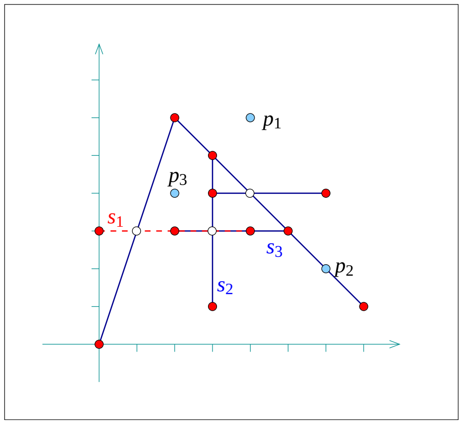


Figure 32.27: An arrangement with history as constructed in *curve_history.cpp*. Note that s_1 and s_3 overlap over two edges. The point-location query points are drawn as lightly shaded dots.

As we are able to keep track of all edges induced by an input curve, we also provide a free function that removes a curve from an arrangement. By calling *remove(arr,ch)*, where *ch* is a valid curve handle, the given curve is deleted from the curve container, and all edges induced solely by this curve (i.e., excluding overlapping edges) are removed from the arrangement. The function returns the number of edges that have been removed.

In some cases, users may need to operate directly on the arrangement edges. We first mention that the specialized insertion functions (see Section 32.2.3) are not supported, as they accept x-monotone curves. Insertion can only be performed via the free insertion-functions. The other edge-manipulation functions (see Section 32.2.3) are however available, but have a different interface that does not use x-monotone curves:

- Invoking *split_edge(e,p)* splits the edge e at a given point p that lies in its interior.
- Invoking *merge_edge(e1,e2)* merges the two given edges. There is a precondition that *e1* and *e2* shared a common end-vertex of degree 2, and that the x-monotone subcurves associated with these edges are mergeable.
- It is possible to remove an edge by simply invoking *remove_edge(e)*.

In all cases, the maintenance of cross-pointers for the appropriate input curves will be done automatically.

It should be noted that it is possible to attach observers to an arrangement-with-history instance in order to get detailed notifications of the changes the arrangements undergoes (see Section 32.7 for the details).

32.10.3 Examples

In the following example we construct a simple arrangement of six line segments, as depicted in Figure 32.27, while maintaining the curve history. The example demonstrates the usage of the special traversal functions. It also shows how to issue point-location queries on the resulting arrangement, using the auxiliary function *point_location_query()* defined in the header file *point_location_utils.h* (see also Section 32.3.1).

```
#include "arr_rational_nt.h"
#include <CGAL/Cartesian.h>
#include <CGAL/Arr_segment_traits_2.h>
```

```
#include <CGAL/Arrangement_on_surface_with_history_2.h>
#include <CGAL/Arrangement_with_history_2.h>
#include <CGAL/Arr_simple_point_location.h>

#include "point_location_utils.h"

typedef CGAL::Cartesian<Number_type>                          Kernel;
typedef CGAL::Arr_segment_traits_2<Kernel>                    Traits_2;
typedef Traits_2::Point_2                                     Point_2;
typedef Traits_2::Curve_2                                     Segment_2;
typedef CGAL::Arrangement_with_history_2<Traits_2>
   Arr_with_hist_2;
typedef Arr_with_hist_2::Curve_handle                         Curve_handle;
typedef CGAL::Arr_simple_point_location<Arr_with_hist_2>  Point_location;

int main()
{
  Arr_with_hist_2   arr;

  // Insert s1, s2 and s3 incrementally:
  Segment_2 s1(Point_2(0, 3), Point_2(4, 3));
  insert(arr, s1);
  Segment_2 s2(Point_2(3, 2), Point_2(3, 5));
  insert(arr, s2);
  Segment_2 s3(Point_2(2, 3), Point_2(5, 3));
  insert(arr, s3);

  // Insert three additional segments aggregately:
  Segment_2 segs[3];
  segs[0] = Segment_2(Point_2(2, 6), Point_2(7, 1));
  segs[1] = Segment_2(Point_2(0, 0), Point_2(2, 6));
  segs[2] = Segment_2(Point_2(3, 4), Point_2(6, 4));
  insert(arr, segs, segs + 3);

  // Print out the curves and the number of edges each one induces.
  Arr_with_hist_2::Curve_iterator            cit;

  std::cout << "The arrangement contains "
            << arr.number_of_curves() << " curves:" << std::endl;
  for (cit = arr.curves_begin(); cit != arr.curves_end(); ++cit)
    std::cout << "Curve [" << *cit << "] induces "
              << arr.number_of_induced_edges(cit) << " edges." <<
   std::endl;

  // Print the arrangement edges, along with the list of curves that
  // induce each edge.
  Arr_with_hist_2::Edge_iterator                  eit;
  Arr_with_hist_2::Originating_curve_iterator     ocit;

  std::cout << "The arrangement is comprised of "
            << arr.number_of_edges() << " edges:" << std::endl;
  for (eit = arr.edges_begin(); eit != arr.edges_end(); ++eit) {
    std::cout << "[" << eit->curve() << "]. Originating curves: ";
```

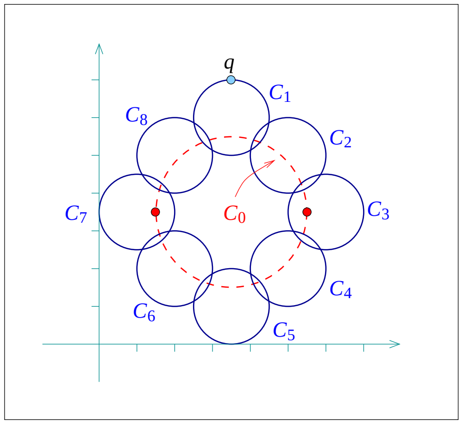


Figure 32.28: An arrangement with history of nine circle as constructed in *edge_manipulation_curve_history.cpp*. Note the vertical tangency points of C_0, marked as dark dots, which subdivide this circle into an upper half and a lower half, each consists of 9 edges. The large circle C_0 is eventually removed from the arrangement, with all 18 edges it induces.

```
    for (ocit = arr.originating_curves_begin (eit);
         ocit != arr.originating_curves_end (eit); ++ocit)
    {
      std::cout << " [" << *ocit << "]" << std::flush;
    }
    std::cout << std::endl;
  }

  // Perform some point-location queries:
  Point_location   pl(arr);

  Point_2          p1(4, 6);
  point_location_query(pl, p1);
  Point_2          p2(6, 2);
  point_location_query(pl, p2);
  Point_2          p3(2, 4);
  point_location_query(pl, p3);

  return 0;
}
```

File: examples/Arrangement_on_surface_2/curve_history.cpp

The following example demonstrates the usage of the free *remove()* function. We construct an arrangement of nine circles, while keeping a handle to each inserted circle. We then remove the large circle C_0, which induces 18 edges, as depicted in Figure 32.28. The example also shows how to use the *split_edge()* and *merge_edge()* functions when operating on an arrangement-with-history instance:

```
#include "arr_rational_nt.h"
#include <CGAL/Cartesian.h>
#include <CGAL/Arr_circle_segment_traits_2.h>
```

```
#include <CGAL/Arrangement_with_history_2.h>

typedef CGAL::Cartesian<Number_type>                    Kernel;
typedef Kernel::Point_2                                 Rat_point_2;
typedef Kernel::Circle_2                                Circle_2;
typedef CGAL::Arr_circle_segment_traits_2<Kernel>       Traits_2;
typedef Traits_2::Point_2                               Point_2;
typedef Traits_2::Curve_2                               Curve_2;
typedef CGAL::Arrangement_with_history_2<Traits_2>      Arr_with_hist_2;
typedef Arr_with_hist_2::Curve_handle                   Curve_handle;
typedef CGAL::Arr_walk_along_line_point_location<Arr_with_hist_2>
                                                        Point_location;

int main ()
{
  // Construct an arrangement containing nine circles: C[0] of radius 2
   and
  // C[1], ..., C[8] of radius 1.
  const Number_type _7_halves = Number_type (7, 2);
  Arr_with_hist_2   arr;
  Curve_2           C[9];
  Curve_handle      handles[9];
  int               k;

  C[0] = Circle_2 (Rat_point_2 (_7_halves, _7_halves), 4,
   CGAL::CLOCKWISE);
  C[1] = Circle_2 (Rat_point_2 (_7_halves, 6), 1, CGAL::CLOCKWISE);
  C[2] = Circle_2 (Rat_point_2 (5, 6), 1, CGAL::CLOCKWISE);
  C[3] = Circle_2 (Rat_point_2 (6, _7_halves), 1, CGAL::CLOCKWISE);
  C[4] = Circle_2 (Rat_point_2 (5, 2), 1, CGAL::CLOCKWISE);
  C[5] = Circle_2 (Rat_point_2 (_7_halves, 1), 1, CGAL::CLOCKWISE);
  C[6] = Circle_2 (Rat_point_2 (2, 2), 1, CGAL::CLOCKWISE);
  C[7] = Circle_2 (Rat_point_2 (1, _7_halves), 1, CGAL::CLOCKWISE);
  C[8] = Circle_2 (Rat_point_2 (2, 5), 1, CGAL::CLOCKWISE);

  for (k = 0; k < 9; k++)
    handles[k] = insert (arr, C[k]);

  std::cout << "The initial arrangement size:" << std::endl
            << "   V = " << arr.number_of_vertices()
            << ",  E = " << arr.number_of_edges()
            << ",  F = " << arr.number_of_faces() << std::endl;

  // Remove the large circle C[0].
  std::cout << "Removing C[0] : ";
  std::cout << remove_curve (arr, handles[0])
            << " edges have been removed." << std::endl;

  std::cout << "The arrangement size:" << std::endl
            << "   V = " << arr.number_of_vertices()
            << ",  E = " << arr.number_of_edges()
            << ",  F = " << arr.number_of_faces() << std::endl;

  // Locate the point q, which should be on an edge e.
```

```
  Point_location                             pl (arr);
  const Point_2                              q = Point_2 (_7_halves, 7);
  CGAL::Object                               obj = pl.locate (q);
  Arr_with_hist_2::Halfedge_const_handle     e;

  CGAL_assertion_code(bool success = ) CGAL::assign (e, obj);
  CGAL_assertion (success);

  // Split the edge e to two edges e1 and e2;
  Arr_with_hist_2::Halfedge_handle           e1, e2;

  e1 = arr.split_edge (arr.non_const_handle (e), q);
  e2 = e1->next();

  std::cout << "After edge split: "
            << "V = " << arr.number_of_vertices()
            << ",  E = " << arr.number_of_edges()
            << ",  F = " << arr.number_of_faces() << std::endl;

  // Merge back the two split edges.
  arr.merge_edge (e1, e2);

  std::cout << "After edge merge: "
            << "V = " << arr.number_of_vertices()
            << ",  E = " << arr.number_of_edges()
            << ",  F = " << arr.number_of_faces() << std::endl;
  return 0;
}
```

File: examples/Arrangement_on_surface_2/edge_manipulation_curve_history.cpp

32.11 Input/Output Streams

In some cases, one would like to save an arrangement object constructed by some application, so that later on it can be restored. In other cases one would like to create nice drawings that represent arrangements constructed by some application. These drawings can be hard printed or displayed on a computer screen.

32.11.1 Input/Output Stream

Consider an arrangement that represents a very complicated geographical map, and assume that there are various applications that need to answer point-location queries on this map. Naturally, you can store the set of curves that induces the arrangement, but this implies that you would need to construct the arrangement from scratch each time you need to reuse it. A more efficient solution is to write the arrangement to a file in a format that other applications can read.

This package provides an *inserter* (the << operator) and an *extractor* (the >> operator) for the *Arrangement_2<Traits,Dcel>* class that inserts and an arrangement object into an output stream and extracts an arrangement object from an input stream respectively. The arrangement is written using a simple predefined ASCII format that encodes the arrangement topology, as well as all geometric entities associated with vertices and edges.

The ability to use the input/output operators, requires that the *Point_2* type and the *X_monotone_curve_2* type defined by the traits class both support the << and >> operators. The *Arr_conic_traits_2* class (see Section 32.6.5), the *Arr_rational_function_traits_2* class (see Section 32.6.6), and the *Arr_linear_traits_2* class (see Section 32.6.2) currently do not provide these operators for the geometric types they define. Thus, only arrangements of line segments or of polylines can be written or read.

The following example constructs the arrangement depicted in Figure 32.7 and writes it to an output file. It also demonstrates how to re-read the arrangement from a file:

```
#include "arr_rational_nt.h"
#include <CGAL/Cartesian.h>
#include <CGAL/Arr_segment_traits_2.h>
#include <CGAL/Arrangement_2.h>
#include <CGAL/IO/Arr_iostream.h>
#include <fstream>

#include "point_location_utils.h"

typedef CGAL::Cartesian<Number_type>                   Kernel;
typedef CGAL::Arr_segment_traits_2<Kernel>             Traits_2;
typedef CGAL::Arrangement_2<Traits_2>                  Arrangement_2;

int main ()
{
  // Construct the arrangement.
  Arrangement_2    arr;

  construct_segments_arr (arr);

  std::cout << "Writing an arrangement of size:" << std::endl
            << "   V = " << arr.number_of_vertices()
            << ",  E = " << arr.number_of_edges()
            << ",  F = " << arr.number_of_faces() << std::endl;

  // Write the arrangement to a file.
  std::ofstream    out_file ("arr_ex_io.dat");

  out_file << arr;
  out_file.close();

  // Read the arrangement from the file.
  Arrangement_2    arr2;
  std::ifstream    in_file ("arr_ex_io.dat");

  in_file >> arr2;
  in_file.close();

  std::cout << "Read an arrangement of size:" << std::endl
            << "   V = " << arr2.number_of_vertices()
            << ",  E = " << arr2.number_of_edges()
            << ",  F = " << arr2.number_of_faces() << std::endl;

  return (0);
}
```

File: examples/Arrangement_on_surface_2/io.cpp

advanced

32.11.2 Arrangements with Auxiliary Data

The inserter and extractor both ignore any auxiliary data stored with the arrangement features. Thus, they are ideal for arrangements instantiated using the *Arr_default_dcel* class. However, as explained in Section 32.8, one can easily extend the arrangement faces by using the *Arr_face_extended_dcel* template, or extend all DCEL records by using the *Arr_extended_dcel* template. In such cases, it might be crucial that the auxiliary data fields are written to the file and read from there.

The arrangement package includes the free functions *write(arr, os, formatter)*, which writes the arrangement *arr* to an output stream *os*, and *read(arr, os, formatter)*, which reads the arrangement *arr* from an input stream *is*. Both operations are performed using a *formatter* object, which defines the I/O format. The package contains three formatter classes:

- *Arr_text_formatter<Arrangement>* defines a simple textual I/O format for the arrangement topology and geometry, disregarding any auxiliary data that may be associated with the arrangement features. This is the default formatter used by the arrangement inserter and the arrangement extractor, as defined above.
- *Arr_face_extended_text_formatter<Arrangement>* operates on arrangements whose DCEL representation is based on the *Arr_face_extended_dcel<Traits,FaceData>* class (see Section 32.8.1). It supports reading and writing the auxiliary data objects stored with the arrangement faces provided that the *FaceData* class supports an inserter and an extractor.
- *Arr_extended_dcel_text_formatter<Arrangement>* operates on arrangements whose DCEL representation is based on the *Arr_extended_dcel<Traits,VertexData,HalfedgeData,FaceData>* class (see Section 32.8.2). It supports reading and writing the auxiliary data objects stored with the arrangement vertices, edges and faces, provided that the *VertexData*, *HalfedgeData* and *FaceData* classed all have inserters and extractors.

The following example constructs the same arrangement as the example *dcel_extension* does (see Section 32.8.2), depicted in Figure 32.25, and writes it to an output file. It also demonstrates how to re-read the arrangement from a file:

```
#include "arr_rational_nt.h"
#include <CGAL/Cartesian.h>
#include <CGAL/Arr_segment_traits_2.h>
#include <CGAL/Arr_extended_dcel.h>
#include <CGAL/Arrangement_2.h>
#include <CGAL/IO/Arr_text_formatter.h>
#include <CGAL/IO/Arr_iostream.h>
#include <fstream>

enum Color {BLUE, RED, WHITE};

std::ostream& operator<< (std::ostream& os, const Color& color)
{
  switch (color)
  {
```

```
  case BLUE:  os << "BLUE";  break;
  case RED:   os << "RED";   break;
  case WHITE: os << "WHITE"; break;
  default: os << "ERROR!";
  }
  return (os);
}

std::istream& operator>> (std::istream& is, Color& color)
{
  std::string   str;
  is >> str;

  if (str == "BLUE")
    color = BLUE;
  else if (str == "RED")
    color = RED;
  else if (str == "WHITE")
    color = WHITE;

  return (is);
}

typedef CGAL::Cartesian<Number_type>                        Kernel;
typedef CGAL::Arr_segment_traits_2<Kernel>                  Traits_2;
typedef Traits_2::Point_2                                   Point_2;
typedef Traits_2::X_monotone_curve_2                        Segment_2;
typedef CGAL::Arr_extended_dcel<Traits_2,
                                Color, bool, int>           Dcel;
typedef CGAL::Arrangement_2<Traits_2, Dcel>                 Arrangement_2;
typedef CGAL::Arr_extended_dcel_text_formatter<Arrangement_2>  Formatter;

int main ()
{
  // Construct the arrangement containing two intersecting triangles.
  Arrangement_2          arr;

  Segment_2      s1 (Point_2(4, 1), Point_2(7, 6));
  Segment_2      s2 (Point_2(1, 6), Point_2(7, 6));
  Segment_2      s3 (Point_2(4, 1), Point_2(1, 6));
  Segment_2      s4 (Point_2(1, 3), Point_2(7, 3));
  Segment_2      s5 (Point_2(1, 3), Point_2(4, 8));
  Segment_2      s6 (Point_2(4, 8), Point_2(7, 3));

  insert_non_intersecting_curve (arr, s1);
  insert_non_intersecting_curve (arr, s2);
  insert_non_intersecting_curve (arr, s3);
  insert (arr, s4);
  insert (arr, s5);
  insert (arr, s6);

  // Go over all arrangement vertices and set their colors.
  Arrangement_2::Vertex_iterator            vit;
  unsigned int                              degree;
```

```
for (vit = arr.vertices_begin(); vit != arr.vertices_end(); ++vit)
{
  degree = vit->degree();
  if (degree == 0)
    vit->set_data (BLUE);       // Isolated vertex.
  else if (degree <= 2)
    vit->set_data (RED);        // Vertex represents an endpoint.
  else
    vit->set_data (WHITE);      // Vertex represents an intersection
 point.
}

// Go over all arrangement edges and set their flags.
Arrangement_2::Edge_iterator                eit;
bool                                        flag;

for (eit = arr.edges_begin(); eit != arr.edges_end(); ++eit)
{
  // Check if the halfegde has the same diretion as its associated
  // segment. Note that its twin always has an opposite direction.
  flag = (eit->source()->point() == eit->curve().source());
  eit->set_data (flag);
  eit->twin()->set_data (!flag);
}

// Go over all arrangement faces and print their outer boundary and
 indices.
Arrangement_2::Face_iterator                fit;
Arrangement_2::Ccb_halfedge_circulator      curr;
int                                         boundary_size;

for (fit = arr.faces_begin(); fit != arr.faces_end(); ++fit)
{
  boundary_size = 0;
  if (! fit->is_unbounded())
  {
    curr = fit->outer_ccb();
    do
    {
      ++boundary_size;
      ++curr;
    } while (curr != fit->outer_ccb());
  }
  fit->set_data (boundary_size);
}

// Write the arrangement to a file.
std::ofstream     out_file ("arr_ex_dcel_io.dat");
Formatter         formatter;

write (arr, out_file, formatter);
out_file.close();
```

```
  // Read the arrangement from the file.
  Arrangement_2    arr2;
  std::ifstream    in_file ("arr_ex_dcel_io.dat");

  read (arr2, in_file, formatter);
  in_file.close();

  std::cout << "The arrangement vertices: " << std::endl;
  for (vit = arr2.vertices_begin(); vit != arr2.vertices_end(); ++vit)
    std::cout << '(' << vit->point() << ") - " << vit->data() <<
   std::endl;

  return (0);
}
```

File: examples/Arrangement_on_surface_2/dcel_extension_io.cpp

You may develop your own own formatter classes — models of the *ArrangementInputFormatter* and *ArrangementOutputFormatter* concepts, as defined in the Reference Manual. Doing so, you can define other I/O formats, such as an XML-based format or a binary format.

advanced

32.11.3 Arrangements with Curve History

Section 32.10 introduces the *Arrangement_with_history_2<Traits,Dcel>* class, which saves the set of curves inducing an arrangement and maintains the relations between these curves and the edges they induce. Naturally, when reading or writing an arrangement-with-history instance we would like this information to be saved to the output stream or restored from the input stream alongside with the basic arrangement structure.

The arrangement package supplies an inserter and an extractor for the *Arrangement_with_history_2<Traits,Dcel>* class. The arrangement is represented using a simple predefined ASCII format. An object of the *Arrangement_with_history_2<Traits,Dcel>* type can be saved and restored, as long as the *Curve_2* type defined by the traits class — as well as the *Point_2* type and the *X_monotone_curve_2* types — support the << and>> operators.

The following example constructs the same arrangement as example *curve_history* does (see Section 32.10.3), depicted in Figure 32.27, and writes it to an output file. It also demonstrates how to re-read the arrangement-with-history from a file:

```
#include "arr_rational_nt.h"
#include <CGAL/Cartesian.h>
#include <CGAL/Arr_segment_traits_2.h>
#include <CGAL/Arrangement_with_history_2.h>
#include <CGAL/IO/Arr_with_history_iostream.h>
#include <fstream>

typedef CGAL::Cartesian<Number_type>                       Kernel;
typedef CGAL::Arr_segment_traits_2<Kernel>                 Traits_2;
typedef Traits_2::Point_2                                  Point_2;
typedef Traits_2::Curve_2                                  Segment_2;
```

```
typedef CGAL::Arrangement_with_history_2<Traits_2>    Arr_with_hist_2;

int main ()
{
  Arr_with_hist_2   arr;

  // Insert six additional segments aggregately:
  Segment_2         segs[6];
  segs[0] = Segment_2 (Point_2 (2, 6), Point_2 (7, 1));
  segs[1] = Segment_2 (Point_2 (3, 2), Point_2 (3, 5));
  segs[2] = Segment_2 (Point_2 (2, 3), Point_2 (5, 3));
  segs[3] = Segment_2 (Point_2 (2, 6), Point_2 (7, 1));
  segs[4] = Segment_2 (Point_2 (0, 0), Point_2 (2, 6));
  segs[5] = Segment_2 (Point_2 (3, 4), Point_2 (6, 4));
  insert (arr, segs, segs + 6);

  std::cout << "Writing an arrangement of "
            << arr.number_of_curves() << " input segments:" << std::endl
            << "   V = " << arr.number_of_vertices()
            << ",  E = " << arr.number_of_edges()
            << ",  F = " << arr.number_of_faces() << std::endl;

  // Write the arrangement to a file.
  std::ofstream     out_file ("arr_ex_io_hist.dat");

  out_file << arr;
  out_file.close();

  // Read the arrangement from the file.
  Arr_with_hist_2   arr2;
  std::ifstream     in_file ("arr_ex_io_hist.dat");

  in_file >> arr2;
  in_file.close();

  std::cout << "Read an arrangement of "
            << arr2.number_of_curves() << " input segments:" << std::endl
            << "   V = " << arr2.number_of_vertices()
            << ",  E = " << arr2.number_of_edges()
            << ",  F = " << arr2.number_of_faces() << std::endl;

  return (0);
}
```

File: examples/Arrangement_on_surface_2/io_curve_history.cpp

advanced

The arrangement package also includes the free functions *write(arr, os, formatter)* and *read(arr, os, formatter)* that operate on a given arrangement-with-history instance *arr*. Both functions are parameterized by a *formatter* object, which define the I/O format. The package contains a template called, *Arr_with_hist_text_formatter< ArranagmentFormatter>*, which extends an arrangement formatter class (see Section 32.11.2) and defines a simple textual input/output format.

advanced

32.11.4 Output QT-widget Stream

You can display the drawings of arrangements in a graphical window powered by QT interactively. All you need to do is follow the guidelines for handling *Qt_widget* objects, and apply the *inserter*, which inserts an arrangement into a *Qt_widget* stream to complete the drawing.

The ability to use this output operator, requires that the *Point_2* and *X_monotone_curve_2* types defined by the traits class both support the << operator that inserts the respective geometric object into a *Qt_widget* stream. The *Arr_rational_function_traits_2* class (see Section 32.6.6) and the *Arr_linear_traits_2* class (see Section 32.6.2) currently do not provide this operator for the geometric types they define. Thus, only arrangements of line segments, polylines, or conic ards can be written. The << operator for polylines and conic arcs defined in *CGAL/IO/Qt_widget_Polyline_2.h* and *CGAL/IO/Qt_widget_Conic_arc_2.h* must be explicitly included, to insert polylines and conic arcs respectively into *Qt_widget* streams.

32.12 Adapting to BOOST Graphs

BOOST[21] is a collection of portable C++ libraries that extend the Standard Template Library (STL). The BOOST Graph Library (BGL), which one of the libraries in the collection, offers an extensive set of generic graph algorithms parameterized through templates. As our arrangements are embedded as planar graphs, it is only natural to extend the underlying data structure with the interface that the BGL expects, and gain the ability to perform the operations that the BGL supports, such as shortest-path computation. This section describes how apply the graph algorithms implemented in the BGL to *Arrangement_2* instances.

An instance of *Arrangement_2* is adapted to a BOOST graph through the provision of a set of free functions that operate on the arrangement features and conform with the relevant BGL concepts. Besides the straightforward adaptation, which associates a vertex with each DCEL vertex and an edge with each DCEL halfedge, the package also offer a *dual* adaptor, which associates a graph vertex with each DCEL face, such that two vertices are connected, iff there is a DCEL halfedge that connects the two corresponding faces.

32.12.1 The Primal Arrangement Representation

Arrangement instances are adapted to BOOST graphs by specializing the *boost:graph_traits* template for *Arrangement_2* instances. The graph-traits states the graph concepts that the arrangement class models (see below) and defines the types required by these concepts.

In this specialization the *Arrangement_2* vertices correspond to the graph vertices, where two vertices are adjacent if there is at least one halfedge connecting them. More precisely, *Arrangement_2::Vertex_handle* is the graph-vertex type, while *Arrangement_2::Halfedge_handle* is the graph-edge type. As halfedges are directed, we consider the graph to be directed as well. Moreover, as several interior-disjoint x-monotone curves (say circular arcs) may share two common endpoints, inducing an arrangement with two vertices that are connected with several edges, we allow parallel edges in our BOOST graph.

Given an *Arrangement_2* instance, we can efficiently traverse its vertices and halfedges. Thus, the arrangement graph is a model of the concepts *VertexListGraph* and *EdgeListGraph* introduced by the BGL. At the same

[21] See also BOOST's homepage at: `www.boost.org`.

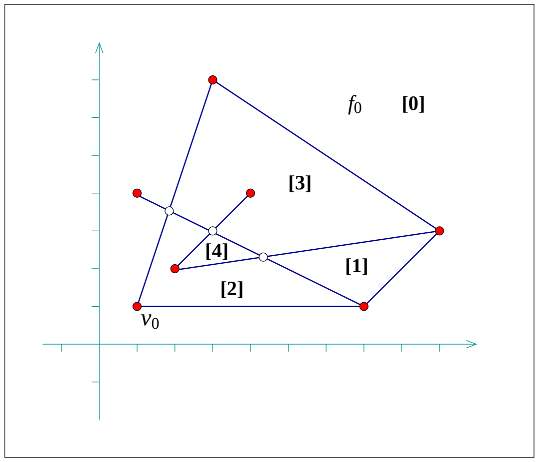


Figure 32.29: An arrangement of 7 line segments, as constructed by *bgl_primal_adapter.cpp* and *bgl_dual_adapter.cpp*. The breadth-first visit times for the arrangement faces, starting from the unbounded face f_0, are shown is brackets.

time, we use an iterator adapter of the circulator over the halfedges incident to a vertex (*Halfedge_around_vertex_circulator* — see Section 32.2.2), so it is possible to go over the ingoing and outgoing edges of a vertex in linear time. Thus, our arrangement graph is a model of the concept *BidirectionalGraph* (this concept refines *IncidenceGraph*, which requires only the traversal of outgoing edges).

It is important to notice that the vertex descriptors we use are *Vertex_handle* objects and *not* vertex indices. However, in order to gain more efficiency in most BGL algorithm, it is better to have them indexed $0, 1, \ldots, (n-1)$, where n is the number of vertices. We therefore introduce the *Arr_vertex_index_map<Arrangement>* class-template, which maintains a mapping of vertex handles to indices, as required by the BGL. An instance of this class must be attached to a valid arrangement vertex when it is created. It uses the notification mechanism (see Section 32.7) to automatically maintain the mapping of vertices to indices, even when new vertices are inserted into the arrangement or existing vertices are removed.

In most algorithm provided by the BGL, the output is given by *property maps*, such that each map entry corresponds to a vertex. For example, when we compute the shortest paths from a given source vertex s to all other vertices we can obtain a map of distances and a map of predecessors — namely for each v vertex we have its distance from s and a descriptor of the vertex that precedes v in the shortest path from s. If the vertex descriptors are simply indices, one can use vectors to efficiently represent the property maps. As this is not the case with the arrangement graph, we offer the *Arr_vertex_property_map<Arrangement,Type>* template allows for an efficient mapping of *Vertex_handle* objects to properties of type *Type*. Note however that unlike the *Arr_vertex_index_map* class, the vertex property-map class is not kept synchronized with the number of vertices in the arrangement, so it should not be reused in calls to BGL functions in case the arrangement is modified in between these calls.

In the following example we construct an arrangement of 7 line segments, as shown in Figure 74.1, then use Dijkstra's shortest-paths algorithm from the BGL to compute the graph distance of all vertices from the leftmost vertex in the arrangement v_0. Note the usage of the *Arr_vertex_index_map* and the *Arr_vertex_property_map* classes. The latter one, instantiated by the type *double* is used to map vertices to their distances from v_0.

```
#include "arr_rational_nt.h"
#include <CGAL/Cartesian.h>
#include <CGAL/Arr_segment_traits_2.h>
#include <CGAL/Arrangement_2.h>
#include <CGAL/graph_traits_Arrangement_2.h>
```

```
#include <CGAL/Arr_vertex_index_map.h>

#include <climits>
#include <boost/graph/dijkstra_shortest_paths.hpp>

typedef CGAL::Cartesian<Number_type>                    Kernel;
typedef CGAL::Arr_segment_traits_2<Kernel>              Traits_2;
typedef Traits_2::Point_2                               Point_2;
typedef Traits_2::X_monotone_curve_2                    Segment_2;
typedef CGAL::Arrangement_2<Traits_2>                   Arrangement_2;

// A functor used to compute the length of an edge.
class Edge_length_func
{
public:

  // Boost property type definitions:
  typedef boost::readable_property_map_tag        category;
  typedef double                                  value_type;
  typedef value_type                              reference;
  typedef Arrangement_2::Halfedge_handle          key_type;

  double operator()(Arrangement_2::Halfedge_handle e) const
  {
    const double     x1 = CGAL::to_double (e->source()->point().x());
    const double     y1 = CGAL::to_double (e->source()->point().y());
    const double     x2 = CGAL::to_double (e->target()->point().x());
    const double     y2 = CGAL::to_double (e->target()->point().y());
    const double     diff_x = x2 - x1;
    const double     diff_y = y2 - y1;

    return std::sqrt(diff_x*diff_x + diff_y*diff_y);
  }
};

double get(Edge_length_func edge_length, Arrangement_2::Halfedge_handle
   e)
{
  return edge_length(e);
}

/* The folowing is a workaround for a bug in the BGL upto and including
   version
 * 103400.
 *
 * Unfortunately some of the calls to the get() function below from the
   BGL
 * code are qualified with the boost namespace, while others are not. For
    The
 * qualified calls the compiler naturally looks for the definition of the
 * function in boost namespace. For the other calls it searches the CGAL
 * namespace according to ADL (Koenig Lookup), as the type of the 1st
 * parameter is in CGAL namespace.
 *
```

```
 * One way to get around it is to provide 2 similar functions that do the
 * same thing. One in CGAL namespace provided in CGAL/Arr_vertex_map.h,
   and
 * the other in boost namespace below. The signature of the latter is
   slightly
 * changed to avoid redefinition. The type of its 1st parameter is
   defined in
 * boost namespace, and is a simple derivation of the 1st parameter of
   the
 * CGAL::get() function.
 */

namespace boost {

template <typename Arrangement_2>
class Arr_vertex_index_map_boost :
    public CGAL::Arr_vertex_index_map<Arrangement_2>
{
 public:
  typedef CGAL::Arr_vertex_index_map<Arrangement_2>     Base;
  /*! Default constructor. */
  Arr_vertex_index_map_boost() : Base() {}

  /*! Constructor from CGAL index map. */
  Arr_vertex_index_map_boost(Base & other) :
    CGAL::Arr_vertex_index_map<Arrangement_2>(other)
  {}
};

/*!
 * Get the index property-map function. Provided so that boost is able to
 * access the Arr_vertex_index_map above.
 * \param index_map The index map.
 * \param v A vertex handle.
 * \return The vertex index.
 */
template<class Arrangement>
unsigned int
get(const boost::Arr_vertex_index_map_boost<Arrangement> & index_map,
    typename Arrangement::Vertex_handle v)
{
  const CGAL::Arr_vertex_index_map<Arrangement> & index_map_tmp =
    static_cast<const CGAL::Arr_vertex_index_map<Arrangement>
   &>(index_map);
  return CGAL::get<Arrangement>(index_map_tmp, v);
}

}

int main()
{
  Arrangement_2   arr;

  // Construct an arrangement of seven intersecting line segments.
```

```
  // We keep a handle for the vertex v_0 that corresponds to the point
    (1,1).
  Arrangement_2::Halfedge_handle  e =
    insert_non_intersecting_curve (arr, Segment_2 (Point_2 (1, 1),
                                                   Point_2 (7, 1)));
  Arrangement_2::Vertex_handle    v0 = e->source();
  insert (arr, Segment_2 (Point_2 (1, 1), Point_2 (3, 7)));
  insert (arr, Segment_2 (Point_2 (1, 4), Point_2 (7, 1)));
  insert (arr, Segment_2 (Point_2 (2, 2), Point_2 (9, 3)));
  insert (arr, Segment_2 (Point_2 (2, 2), Point_2 (4, 4)));
  insert (arr, Segment_2 (Point_2 (7, 1), Point_2 (9, 3)));
  insert (arr, Segment_2 (Point_2 (3, 7), Point_2 (9, 3)));

  // Create a mapping of the arrangement vertices to indices.
  CGAL::Arr_vertex_index_map<Arrangement_2>        index_map_tmp(arr);
  boost::Arr_vertex_index_map_boost<Arrangement_2>
   index_map(index_map_tmp);

  // Perform Dijkstra's algorithm from the vertex v0.
  Edge_length_func                                      edge_length;
  CGAL::Arr_vertex_property_map<Arrangement_2, double>
   dist_map(index_map);
  boost::dijkstra_shortest_paths(arr, v0,
                                 boost::vertex_index_map(index_map).
                                 weight_map(edge_length).
                                 distance_map(dist_map));

  // Print the results:
  Arrangement_2::Vertex_iterator      vit;

  std::cout << "The distances of the arrangement vertices from ("
            << v0->point() << ") :" << std::endl;
  for (vit = arr.vertices_begin(); vit != arr.vertices_end(); ++vit)
    std::cout << "(" << vit->point() << ") at distance "
              << dist_map[vit] << std::endl;

  return 0;
}
```

File: examples/Arrangement_on_surface_2/bgl_primal_adapter.cpp

32.12.2 The Dual Arrangement Representation

It is possible to give a dual graph representation for an arrangement instance, such that each arrangement face corresponds to a graph vertex and two vertices are adjacent iff the corresponding faces share a common edge on their boundaries. This is done by specializing the *boost:graph_traits* template for *Dual<Arrangement_2>* instances, where *Dual<Arrangement_2>* is a template specialization that gives a dual interpretation to an arrangement instance.

In dual representation, *Arrangement_2::Face_handle* is the graph-vertex type, while *Arrangement_2::Halfedge_handle* is the graph-edge type. We treat the graph edges as directed, such that a halfedge e is directed from f_1, which is its incident face, to f_2, which is the incident face of its twin halfedge. As two arrangement faces may share more than a single edge on their boundary, we allow parallel edges in our BOOST graph. As is the case in the primal graph, the dual arrangement graph is also a model of the concepts *VertexListGraph*, *EdgeListGraph* and *BidirectionalGraph* (thus also of *IncidenceGraph*).

Since we use *Face_handle* objects as the vertex descriptors, we define the *Arr_face_index_map*<*Arrangement*> class-template, which maintains an efficient mapping of face handles to indices. We also provide the template *Arr_face_property_map*<*Arrangement,Type*> for associating arbitrary data with the arrangement faces.

In the following example we construct the same arrangement as in example *bgl_primal_adapter.cpp* (see Figure 74.1), and perform breadth-first search on the graph faces, starting from the unbounded face. We extend the DCEL faces with an unsigned integer, marking the discover time of the face and use a breadth-first-search visitor to obtain these times and update the faces accordingly:

```
#include "arr_rational_nt.h"
#include <CGAL/Cartesian.h>
#include <CGAL/Arr_segment_traits_2.h>
#include <CGAL/Arr_extended_dcel.h>
#include <CGAL/Arrangement_2.h>

#include <climits>
#include <boost/graph/dijkstra_shortest_paths.hpp>

#include <CGAL/graph_traits_Dual_Arrangement_2.h>
#include <CGAL/Arr_face_index_map.h>

#include "arr_print.h"

typedef CGAL::Cartesian<Number_type>                        Kernel;
typedef CGAL::Arr_segment_traits_2<Kernel>                  Traits_2;
typedef Traits_2::Point_2                                   Point_2;
typedef Traits_2::X_monotone_curve_2                        Segment_2;
typedef CGAL::Arr_face_extended_dcel<Traits_2,
                                     unsigned int>          Dcel;
typedef CGAL::Arrangement_2<Traits_2, Dcel>                 Arrangement_2;
typedef CGAL::Dual<Arrangement_2>
   Dual_arrangement_2;

// A BFS visitor class that associates each vertex with its discover
   time.
// In our case graph vertices represent arrangement faces.
template <class IndexMap>
class Discover_time_bfs_visitor : public boost::default_bfs_visitor
{
private:

  const IndexMap  *index_map;      // Mapping vertices to indices.
  unsigned int     time;           // The current time stamp.

public:

  // Constructor.
  Discover_time_bfs_visitor (const IndexMap& imap) :
    index_map (&imap),
```

```
    time (0)
  {}

  // Write the discover time for a given vertex.
  template <typename Vertex, typename Graph>
  void discover_vertex(Vertex u, const Graph& /* g */)
  {
    u->set_data (time);
    time++;
  }
};

int main ()
{
  Arrangement_2   arr;

  // Construct an arrangement of seven intersecting line segments.
  insert (arr, Segment_2 (Point_2 (1, 1), Point_2 (7, 1)));
  insert (arr, Segment_2 (Point_2 (1, 1), Point_2 (3, 7)));
  insert (arr, Segment_2 (Point_2 (1, 4), Point_2 (7, 1)));
  insert (arr, Segment_2 (Point_2 (2, 2), Point_2 (9, 3)));
  insert (arr, Segment_2 (Point_2 (2, 2), Point_2 (4, 4)));
  insert (arr, Segment_2 (Point_2 (7, 1), Point_2 (9, 3)));
  insert (arr, Segment_2 (Point_2 (3, 7), Point_2 (9, 3)));

  // Create a mapping of the arrangement faces to indices.
  CGAL::Arr_face_index_map<Arrangement_2>      index_map (arr);

  // Perform breadth-first search from the unbounded face, and use the
   BFS
  // visitor to associate each arrangement face with its discover time.
  Discover_time_bfs_visitor<CGAL::Arr_face_index_map<Arrangement_2> >
                                               bfs_visitor (index_map);
  Arrangement_2::Face_handle                   uf = arr.unbounded_face();

  boost::breadth_first_search (Dual_arrangement_2 (arr), uf,
                               boost::vertex_index_map (index_map).
                               visitor (bfs_visitor));

  // Print the results:
  Arrangement_2::Face_iterator      fit;

  for (fit = arr.faces_begin(); fit != arr.faces_end(); ++fit) {
    std::cout << "Discover time " << fit->data() << " for ";
    if (fit != uf) {
      std::cout << "face ";
      print_ccb<Arrangement_2> (fit->outer_ccb());
    }
    else
      std::cout << "the unbounded face." << std::endl;
  }

  return 0;
}
```

File: `examples/Arrangement_on_surface_2/bgl_dual_adapter.cpp`

32.13 How To Speed Up Your Computation

Before the specific tips, we remind you that compiling programs with debug flags disabled and with optimization flags enabled significantly reduces the running time.

1. When the curves to be inserted into an arrangement are x-monotone and pairwise disjoint in their interior to start with, then it is more efficient (in running time) and less demanding (in traits-class functionality) to use the non-intersection insertion-functions instead of the general ones; e.g., *insert()*.

2. When the curves to be inserted into an arrangement are segments that are pairwise disjoint in their interior, it is more efficient to use the traits class *Arr_non_caching_segment_traits_2* rather then the default one (*Arr_segment_traits_2*).

   If the segments may intersect each other, the default traits class *Arr_segment_traits_2* can be safely used with the somehow limited number type *Quotient<MP_float>*.

   On rare occasions the traits class *Arr_non_caching_segment_traits_2* exhibits slightly better performance than the default one (*Arr_segment_traits_2* even when the segments intersect each other, due to the small overhead of the latter (optimized) traits class. (For example, when the so called LEDA rational kernel is used).

3. Prior knowledge of the combinatorial structure of the arrangement can be used to accelerate operations that insert x-monotone curves, whose interior is disjoint from existing edges and vertices of the arrangement. The specialized insertion functions, i.e., *insert_in_face_interior()*, *insert_from_left_vertex()*, *insert_from_right_vertex()*, and *insert_at_vertices()* can be used according to the available information. These functions hardly involve any geometric operations, if at all. They accept topologically related parameters, and use them to operate directly on the DCEL records, thus saving algebraic operations, which are especially expensive when high-degree curves are involved.

   A polygon, represented by a list of segments along its boundary, can be inserted into an empty arrangement as follows. First, one segment is inserted using *insert_in_face_interior()* into the unbounded face. Then, a segment with a common end point is inserted using either *insert_from_left_vertex()* or *insert_from_right_vertex()*, and so on with the rest of the segments except for the last, which is inserted using *insert_at_vertices()*, as both endpoints of which are the mapping of known vertices.

4. The main trade-off among point-location strategies, is between time and storage. Using the naive or walk strategies, for example, takes more query time but does not require preprocessing or maintenance of auxiliary structures and saves storage space.

5. If point-location queries are not performed frequently, but other modifying functions, such as removing, splitting, or merging edges are, then using a point-location strategy that does not require the maintenance of auxiliary structures, such as the naive or walk strategies, is preferable.

6. There is a trade-off between two modes of the trapezoidal RIC strategy that enables the user to choose whether preprocessing should be performed or not. If preprocessing is not used, the creation of the structure is faster. However, for some input sequences the structure might be unbalanced and therefore queries and updates might take longer, especially, if many removal and split operations are performed.

7. When the curves to be inserted into an arrangement are available in advance (as opposed to supplied on-line), it is advised to use the more efficient aggregate (sweep-based) insertion over the incremental insertion; e.g., *insert()*.

8. The various traits classes should be instantiated with an exact number type to ensure robustness, when the input of the operations to be carried out might be degenerate, although inexact number types could be used at the user's own risk.

9. Maintaining short bit-lengths of coordinate representations may drastically decrease the time consumption of arithmetic operations on the coordinates. This can be achieved by caching certain information or normalization (of rational numbers). However, both solutions should be used cautiously, as the former may lead to an undue space consumption, and indiscriminate normalization may considerably slow down the overall process.

10. Geometric functions (e.g., traits methods) dominate the time consumption of most operations. Thus, calls to such function should be avoided or at least their number should be decreased, perhaps at the expense of increased combinatorial-function calls or increased space consumption. For example, repetition of geometric-function calls could be avoided by storing the results obtained by the first call, and reusing them when needed.

Design and Implementation History

The code of this package is the result of a long development process. Initially (and until version 3.1), the code was spread among several components, namely, *Topological_map*, *Planar_map_2*, *Planar_map_with_intersections_2* and *Arrangement_2*, that were developed by Ester Ezra, Eyal Flato, Efi Fogel, Dan Halperin, Iddo Hanniel, Idit Haran, Shai Hirsch, Eugene Lipovetsky, Oren Nechushtan, Sigal Raab, Ron Wein, Baruch Zukerman, and Tali Zvi.

In version 3.2, as part of the ACS project, the packages have gone through a major re-design, resulting in an improved and unified *2D Arrangements* package. The code of the new package was restructured and developed by Efi Fogel, Idit Haran, Ron Wein, and Baruch Zukerman. This version included for the first time a new geometry-traits class that handles circular and linear curves, and is based on the circular kernel. The circular kernel was developed by Monique Teillaud, Sylvain Pion, and Julien Hazebrouck.

Version 3.3 features arrangements of unbounded curves for the first time. The design and development of this feature required yet another restructuring of the entire package. All this was done by Eric Berberich, Efi Fogel, Dan Halperin, Ophir Setter, and Ron Wein. Michael Hemmer helped tuning up parts of the geometry-traits concept related to unbounded curves.

Version 3.7 introduced a geometry-traits class that handles planar algebraic curves of arbitrary degree. It was developed by Eric Berberich and Michael Kerber.

Version 3.9 introduced a new geometry-traits class that handles rational arcs. It was developed by Oren Salzman and Michael Hemmer. It replaced an old traits, which handled the same family of curves, developed by Ron Wein.

Version 4.1 introduces a revised implementation of the point location class via a randomized incremental construction of the trapezoidal map. The old class was implemented by Oren Nechushtan, while the revamp was done by Michal Kleinbort and Michael Hemmer. The new class adds support for unbounded curves and can now guarantee logarithmic query time in all cases.

2D Arrangements
Reference Manual

Ron Wein, Efi Fogel, Baruch Zukerman, Dan Halperin, Eric Berberich, and Oren Zalzman

Given a set $\mathcal{C}$ of planar curves, the *arrangement* $\mathcal{A}(\mathcal{C})$ is the subdivision of the plane induced by the curves in $\mathcal{C}$ into maximally connected cells. The cells can be 0-dimensional (*vertices*), 1-dimensional (*edges*) or 2-dimensional (*faces*).

The class *Arrangement_2<Traits,Dcel>* encapsulates a data structure that maintains arrangements of arbitrary bounded planar curves. It comes with a variety of algorithms that operate on planar arrangement, such as point-location queries and overlay computations, which are implemented as peripheral classes or as free (global) functions.

32.14 Classified Reference Pages

Enumerations

Tags

Concepts

Geometric Object Concepts

Function Object Concepts

Classes

Tags

Functions

32.15 Alphabetical List of Reference Pages

CGAL::Arrangement_2<Traits,Dcel>

Definition

An object *arr* of the class *Arrangement_2<Traits,Dcel>* represents the planar subdivision induced by a set of x-monotone curves and isolated points into maximally connected cells. The arrangement is represented as a doubly-connected edge-list (DCEL) such that each DCEL vertex is associated with a point of the plane and each edge is associated with an x-monotone curve whose interior is disjoint from all other edges and vertices. Recall that an arrangement edge is always comprised of a pair of twin DCEL halfedges.

The *Arrangement_2<Traits,Dcel>* template has two parameters:

- The *Traits* template-parameter should be instantiated with a model of the *ArrangementBasicTraits_2* concept. The traits class defines the types of x-monotone curves and two-dimensional points, namely *X_monotone_curve_2* and *Point_2*, respectively, and supports basic geometric predicates on them.
- The *Dcel* template-parameter should be instantiated with a class that is a model of the *ArrangementDcel* concept. The value of this parameter is by default *Arr_default_dcel<Traits>*.

The available traits classes and DCEL classes are described below.

#include <CGAL/Arrangement_2.h>

Types

typedef Arrangement_2<Traits_2,Dcel>	*Self;*	a private type used as an abbreviation of the *Arrangement_2<Traits,Dcel>* type hereafter.
Arrangement_2<Traits,Dcel>:: Traits_2		the traits class in use.
Arrangement_2<Traits,Dcel>:: Dcel		the DCEL representation of the arrangement.
typedef typename Traits_2::Point_2	*Point_2;*	the point type, as defined by the traits class.
typedef typename Traits_2::X_monotone_curve_2	*X_monotone_curve_2;*	the x-monotone curve type, as defined by the traits class.
typedef typename Dcel::Size	*Size;*	the size type (equivalent to *size_t*).

Arrangement_2<Traits,Dcel>:: Vertex	represents a 0-dimensional cell in the subdivision. A vertex is always associated with a point.
Arrangement_2<Traits,Dcel>:: Halfedge	represents (together with its twin — see below) a 1-dimensional cell in the subdivision. A halfedge is always associated with an x-monotone curve.
Arrangement_2<Traits,Dcel>:: Face	represents a 2-dimensional cell in the subdivision.

The following handles, iterators, and circulators all have respective constant counterparts (for example, in addition to *Vertex_iterator* the type *Vertex_const_iterator* is also defined). See [MS96] for a discussion of constant versus mutable iterator types. The mutable types are assignable to their constant counterparts.

Vertex_iterator, *Halfedge_iterator*, and *Face_iterator* are equivalent to the respective handle types (namely, *Vertex_handle*, *Halfedge_handle*, and *Face_handle*). Thus, wherever the handles appear in function parameter lists, the respective iterators can be passed as well.

Arrangement_2<Traits,Dcel>:: Vertex_handle	a handle for an arrangement vertex.
Arrangement_2<Traits,Dcel>:: Halfedge_handle	a handle for a halfedge. The halfedge and its twin form together an arrangement edge.
Arrangement_2<Traits,Dcel>:: Face_handle	a handle for an arrangement face.
Arrangement_2<Traits,Dcel>:: Vertex_iterator	a bidirectional iterator over the vertices of the arrangement. Its value-type is *Vertex*.
Arrangement_2<Traits,Dcel>:: Halfedge_iterator	a bidirectional iterator over the halfedges of the arrangement. Its value-type is *Halfedge*.
Arrangement_2<Traits,Dcel>:: Edge_iterator	a bidirectional iterator over the edges of the arrangement. (That is, it skips every other halfedge.) Its value-type is *Halfedge*.
Arrangement_2<Traits,Dcel>:: Face_iterator	a bidirectional iterator over the faces of arrangement. Its value-type is *Face*.
Arrangement_2<Traits,Dcel>:: Unbounded_face_iterator	a bidirectional iterator over the unbounded faces of arrangement. Its value-type is *Face*.
Arrangement_2<Traits,Dcel>:: Halfedge_around_vertex_circulator	a bidirectional circulator over the halfedges that have a given vertex as their target. Its value-type is *Halfedge*.
Arrangement_2<Traits,Dcel>:: Ccb_halfedge_circulator	a bidirectional circulator over the halfedges of a CCB (connected component of the boundary). Its value-type is *Halfedge*. Each bounded face has a single CCB representing it outer boundary, and may have several inner CCBs representing its holes.
Arrangement_2<Traits,Dcel>:: Hole_iterator	a bidirectional iterator over the holes (i.e., inner CCBs) contained inside a given face. Its value type is *Ccb_halfedge_circulator*.
Arrangement_2<Traits,Dcel>:: Isolated_vertex_iterator	a bidirectional iterator over the isolated vertices contained inside a given face. Its value type is *Vertex*.

Creation

Arrangement_2<Traits,Dcel> arr;	constructs an empty arrangement containing one unbounded face, which corresponds to the entire plane.

Arrangement_2<Traits,Dcel> arr(Self other); copy constructor.

*Arrangement_2<Traits,Dcel> arr(const Traits_2 *traits);*

constructs an empty arrangement that uses the given *traits* instance for performing the geometric predicates.

Assignment Methods

Self&	*arr = other*	assignment operator.
void	*arr.assign(Self other)*	assigns the contents of another arrangement.
void	*arr.clear()*	clears the arrangement.

Access Functions

*Traits_2**	*arr.get_traits()*	returns the traits object used by the arrangement instance. A *const* version is also available.
bool	*arr.is_empty() const*	determines whether the arrangement is empty (contains only the unbounded face, with no vertices or edges).

All *_begin()* and *_end()* methods listed below also have *const* counterparts, returning constant iterators instead of mutable ones:

• *Accessing the Arrangement Vertices:*

Size *arr.number_of_vertices() const*

returns the number of vertices in the arrangement.

Size *arr.number_of_isolated_vertices() const*

returns the total number of isolated vertices in the arrangement.

Vertex_iterator	*arr.vertices_begin()*	returns the begin-iterator of the vertices in the arrangement.
Vertex_iterator	*arr.vertices_end()*	returns the past-the-end iterator of the vertices in the arrangement.

Size *arr.number_of_vertices_at_infinity() const*

returns the number of arrangement vertices that lie at infinity and are not associated with valid points. Such vertices are not considered to be regular arrangement vertices and *arr.number_of_vertices()* does not count them.

• *Accessing the Arrangement Edges:*

Size	*arr.number_of_halfedges() const*	
		returns the number of halfedges in the arrangement.
Halfedge_iterator	*arr.halfedges_begin()*	returns the begin-iterator of the halfedges in the arrangement.
Halfedge_iterator	*arr.halfedges_end()*	returns the past-the-end iterator of the halfedges in the arrangement.
Size	*arr.number_of_edges() const*	
		returns the number of edges in the arrangement (equivalent to *arr.number_of_halfedges() / 2*).
Edge_iterator	*arr.edges_begin()*	returns the begin-iterator of the edges in the arrangement.
Edge_iterator	*arr.edges_end()*	returns the past-the-end iterator of the edges in the arrangement.

• *Accessing the Arrangement Faces:*

Face_handle	*arr.unbounded_face()*	returns a handle for an unbounded face of the arrangement. In case the arrangement comprises only bounded curves, there is a single unbounded face and the function returns a handle to it. Otherwise, a handle to an arbitrary unbounded face is returned.
Size	*arr.number_of_faces() const*	
		returns the number of faces in the arrangement.
Face_iterator	*arr.faces_begin()*	returns the begin-iterator of the faces in the arrangement.
Face_iterator	*arr.faces_end()*	returns the past-the-end iterator of the faces in the arrangement.
Size	*arr.number_of_unbounded_faces() const*	
		returns the number of unbounded faces in the arrangement. Note *arr.number_of_faces()* also counts the unbounded faces of the arrangement.
Unbounded_face_iterator		
	arr.unbounded_faces_begin()	
		returns the begin-iterator of the unbounded faces in the arrangement.

Unbounded_face_iterator

	arr.unbounded_faces_end()	returns the past-the-end iterator of the unbounded faces in the arrangement.
Face_handle	*arr.fictitious_face()*	returns a handle to the fictitious face of the arrangement. If the arrangement is not unbounded, there is no fictitious face. In this case the result is not deterministic. A const version is also available.

advanced

Casting away Constness

It is sometime necessary to convert a constant (non-mutable) handle to a mutable handle. For example, the result of a point-location query is a non-mutable handle for the arrangement cell containing the query point. Assume that the query point lies on a edge, so we obtain a *Halfedge_const_handle*; if we wish to use this handle and remove the edge, we first need to cast away its "constness".

Vertex_handle *arr.non_const_handle(Vertex_const_handle v)*

casts the given constant vertex handle to an equivalent mutable handle.

Halfedge_handle *arr.non_const_handle(Halfedge_const_handle e)*

casts the given constant halfedge handle to an equivalent mutable handle.

Face_handle *arr.non_const_handle(Face_const_handle f)*

casts the given constant face handle to an equivalent mutable handle.

advanced

Modifiers

• *Specialized Insertion Methods:*

Vertex_handle *arr.insert_in_face_interior(Point_2 p, Face_handle f)*

inserts the point p into the arrangement as an isolated vertex in the interior of the face f and returns a handle for the newly created vertex.
Precondition: p lies in the interior of the face f.

Halfedge_handle *arr.insert_in_face_interior(X_monotone_curve_2 c, Face_handle f)*

inserts the curve c that is entirely contained in the interior of a given face f. If c is a bounded curve two new vertices that correspond to c's endpoints are created and connected with a newly created halfedge pair, which forms a new hole (inner component) in the face f. If c is unbounded, at least one of the two vertices that represents its end lies at infinity, and its creation modifies the outer boundary of f. The function returns a handle for one of the new halfedges corresponding to the inserted curve, directed in lexicographic increasing order (from left to right).
Precondition: c lies entirely in the interior of the face f and is disjoint from all existing arrangement vertices and edges (in particular, both its endpoints are not already associated with existing arrangement vertices).
Precondition: In case c is an unbounded curve, f must be an unbounded face.

Halfedge_handle *arr.insert_from_left_vertex(X_monotone_curve_2 c, Vertex_handle v)*

inserts the curve c into the arrangement, such that its left endpoint corresponds to a given arrangement vertex. As a result, a new vertex that correspond to c's right endpoint is created and connected to v with a newly created halfedge pair. If c has an unbounded right end, the new vertex lies at infinity and the unbounded face that contains the interior of the curve is split. The function returns a handle for one of the new halfedges corresponding to the inserted curve, directed towards the newly created vertex — that is, directed in lexicographic increasing order (from left to right).
Precondition: The interior of c is disjoint from all existing arrangement vertices and edges.
Precondition: v is associated with the left endpoint of c.
Precondition: The right endpoint of c is not already associated with an existing arrangement vertex.

Halfedge_handle *arr.insert_from_right_vertex(X_monotone_curve_2 c, Vertex_handle v)*

inserts the curve c into the arrangement, such that its right endpoint corresponds to a given arrangement vertex. As a result, a new vertex that correspond to c's left endpoint is created and connected to v with a newly created halfedge pair. If c has an unbounded left end, the new vertex lies at infinity and the unbounded face that contains the interior of the curve is split. The function returns a handle for one of the new halfedges corresponding to the inserted curve, directed to the newly created vertex — that is, directed in lexicographic decreasing order (from right to left).
Precondition: The interior of c is disjoint from all existing arrangement vertices and edges.
Precondition: v is associated with the right endpoint of c.
Precondition: The left endpoint of c is not already associated with an existing arrangement vertex.

Halfedge_handle *arr.insert_at_vertices(X_monotone_curve_2 c, Vertex_handle v1, Vertex_handle v2)*

inserts the curve *c* into the arrangement, such that both *c*'s endpoints correspond to existing arrangement vertices, given by *v1* and *v2*. The function creates a new halfedge pair that connects the two vertices, and returns a handle for the halfedge directed from *v1* to *v2*.
Precondition: The interior of *c* is disjoint from all existing arrangement vertices and edges.
Precondition: *c* must not be an unbounded curve.
Precondition: *v1* and *v2* are associated with *c*'s endpoints.
Precondition: If *v1* and *v2* are already connected by an edge, this edge represents an *x*-monotone curve that is interior-disjoint from *c*).

advanced

Halfedge_handle *arr.insert_in_face_interior(X_monotone_curve_2 c, Halfedge_handle fict_pred1, Halfedge_handle fict_pred2 = Halfedge_handle())*

inserts an unbounded curve *c* into the arrangement, such that *c* is entirely contained within a single unbounded face of the arrangement. *fict_pred1* specifies the fictitious halfedge that should contain the vertex at infinity that corresponds to the unbounded end of *c*. If both ends of *c* are unbounded, *fict_pred1* indicated the place for its left end and *fict_pred2* indicated a place for its right end. The function returns a handle for one of the new halfedges directed (lexicographically) from left to right.
Precondition: *c* is an unbounded curve disjoint from all existing arrangement vertices and edges.
Precondition: *fict_pred1* (and *fict_pred2*) are fictitious halfedges that contains the unbounded end(s) of *c*. If both halfedges are given they must be both incident to the same unbounded face.

Halfedge_handle *arr.insert_from_left_vertex(X_monotone_curve_2 c, Halfedge_handle pred)*

inserts the curve *c* into the arrangement, such that its left endpoint corresponds to a given arrangement vertex. This vertex is the target vertex of the halfedge *pred*, such that *c* is inserted to the circular list of halfedges around *pred->target()* right between *pred* and its successor. The function returns a handle for one of the new halfedges directed (lexicographically) from left to right.
Precondition: The interior of *c* is disjoint from all existing arrangement vertices and edges.
Precondition: *pred->target()* is associated with the left endpoint of *c*, and *c* should be inserted after *pred* in a clockwise order around this vertex.
Precondition: The right endpoint of *c* is not already associated with an existing arrangement vertex.

Halfedge_handle *arr.insert_from_left_vertex(X_monotone_curve_2 c,*
Halfedge_handle pred,
Halfedge_handle fict_pred)

inserts an unbounded curve c into the arrangement, such that its left endpoint is bounded and corresponds to a given arrangement vertex. This vertex is the target vertex of the halfedge *pred*, such that c is inserted to the circular list of halfedges around *pred->target()* right between *pred* and its successor. Similarly, *fict_pred* specifies the fictitious halfedge that should contain the vertex at infinity that corresponds to the unbounded right end of c. The function returns a handle for one of the new halfedges directed (lexicographically) from left to right.
Precondition: The interior of c is disjoint from all existing arrangement vertices and edges. c must have a bounded left endpoint and an unbounded right end.
Precondition: *pred->target()* is associated with the left endpoint of c, and c should be inserted after *pred* in a clockwise order around this vertex.
Precondition: *fict_pred* is a fictitious halfedge that contains the unbounded right end of c.

Halfedge_handle *arr.insert_from_right_vertex(X_monotone_curve_2 c, Halfedge_handle pred)*

inserts the curve c into the arrangement, such that its right endpoint corresponds to a given arrangement vertex. This vertex is the target vertex of the halfedge *pred*, such that c is inserted to the circular list of halfedges around *pred->target()* right between *pred* and its successor. The function returns a handle for one of the new halfedges directed (lexicographically) from right to left.
Precondition: The interior of c is disjoint from all existing arrangement vertices and edges.
Precondition: *pred->target()* is associated with the right endpoint of c, and c should be inserted after *pred* in a clockwise order around this vertex.
Precondition: The left endpoint of c is not already associated with an existing arrangement vertex.

Halfedge_handle *arr.insert_from_right_vertex(X_monotone_curve_2 c,*
Halfedge_handle pred,

Halfedge_handle fict_pred)

inserts an unbounded curve *c* into the arrangement, such that its right endpoint is bounded and corresponds to a given arrangement vertex. This vertex is the target vertex of the halfedge *pred*, such that *c* is inserted to the circular list of halfedges around *pred->target()* right between *pred* and its successor. Similarly, *fict_pred* specifies the fictitious halfedge that should contain the vertex at infinity that corresponds to the unbounded left end of *c*. The function returns a handle for one of the new halfedges directed (lexicographically) from right to left.
Precondition: The interior of *c* is disjoint from all existing arrangement vertices and edges. *c* must have a bounded right endpoint and an unbounded left end.
Precondition: *pred->target()* is associated with the right endpoint of *c*, and *c* should be inserted after *pred* in a clockwise order around this vertex.
Precondition: *fict_pred* is a fictitious halfedge that contains the unbounded left end of *c*.

Halfedge_handle *arr.insert_at_vertices(X_monotone_curve_2 c,*
Halfedge_handle pred1,
Vertex_handle v2)

inserts the curve *c* into the arrangement, such that both *c*'s endpoints correspond to existing arrangement vertices, given by *pred1->target()* and *v2*. The function creates a new halfedge pair that connects the two vertices (where the corresponding halfedge is inserted right between *pred1* and its successor around *pred1*'s target vertex) and returns a handle for the halfedge directed from *pred1->target()* to *v2*.
Precondition: The interior of *c* is disjoint from all existing arrangement vertices and edges.
Precondition: *pred1->target()* and *v2* are associated with *c*'s endpoints.
Precondition: If *pred1->target* and *v2* are already connected by an edge, this edge represents an *x*-monotone curve that is interior-disjoint from *c*).

Halfedge_handle *arr.insert_at_vertices(X_monotone_curve_2 c,*
Halfedge_handle pred1,

Halfedge_handle pred2)

inserts the curve *c* into the arrangement, such that both *c*'s endpoints correspond to existing arrangement vertices, given by *pred1->target()* and *pred2->target()*. The function creates a new halfedge pair that connects the two vertices (with *pred1* and *pred2* indicate the exact place for these halfedges around the two target vertices) and returns a handle for the halfedge directed from *pred1->target()* to *pred2->target()*.
Precondition: The interior of *c* is disjoint from all existing arrangement vertices and edges.
Precondition: *pred1->target()* and *pred2->target()* are associated with *c*'s endpoints.
Precondition: If *pred1->target* and *pred2->target()* are already connected by an edge, this edge represents an *x*-monotone curve that is interior-disjoint from *c*).

advanced

• *Modifying Vertices and Edges:*

Vertex_handle *arr.modify_vertex(Vertex_handle v, Point_2 p)*

sets *p* to be the point associated with the vertex *v*. The function returns a handle for the modified vertex (same as *v*).
Precondition: *v* is not a vertex at infinity and *p* is geometrically equivalent to the point currently associated with *v*.

Face_handle *arr.remove_isolated_vertex(Vertex_handle v)*

removes the isolated vertex *v* from the arrangement. The function returns the face *f* that used to contain the isolated vertex.
Precondition: *v* is an isolated vertex (has no incident edges).

Halfedge_handle *arr.modify_edge(Halfedge_handle e, X_monotone_curve_2 c)*

sets *c* to be the *x*-monotone curve associated with the edge *e*. The function returns a handle for the modified edge (same as *e*).
Precondition: *c* is geometrically equivalent to the curve currently associated with *e*.

Halfedge_handle *arr.split_edge(Halfedge_handle e, X_monotone_curve_2 c1, X_monotone_curve_2 c2)*

splits the edge *e* into two edges (more precisely, into two halfedge pairs), associated with the given subcurves *c1* and *c2*, and creates a vertex that corresponds to the split point. The function returns a handle for the halfedge, whose source is the same as *e->source()* and whose target vertex is the split point.
Precondition: Either *c1*'s left endpoint and *c2*'s right endpoint correspond to *e*'s end-vertices such that *c1*'s right endpoint and *c2*'s left endpoint are equal and define the split point — or vice-versa (with change of roles between *c1* and *c2*).

Halfedge_handle *arr.merge_edge(Halfedge_handle e1, Halfedge_handle e2, X_monotone_curve_2 c)*

merges the edges represented by *e1* and *e2* into a single edge, associated with the given merged curve *c*. Denote *e1*'s end-vertices as u_1 and v, while *e2*'s end-vertices are denoted u_2 and v. The function removes the common vertex v returns a handle for one of the merged halfedges, directed from u_1 to u_2.
Precondition: *e1* and *e2* share a common end-vertex, such that the two other end-vertices of the two edges are associated with *c*'s endpoints.

Face_handle *arr.remove_edge(Halfedge_handle e, bool remove_source = true, bool remove_target = true)*

removes the edge *e* from the arrangement. Since the *e* may be the only edge incident to its source vertex (or its target vertex), this vertex can be removed as well. The flags *remove_source* and *remove_target* indicate whether the endpoints of *e* should be removed, or whether they should be left as isolated vertices in the arrangement. If the operation causes two faces to merge, the merged face is returned. Otherwise, the face to which the edge was incident is returned.

Miscellaneous

bool *arr.is_valid() const*

returns *true* if *arr* represents a valid instance of *Arrangement_2<Traits,Dcel>*. In particular, the functions checks the topological structure of the arrangement and assures that it is valid. In addition, the function performs several simple geometric tests to ensure the validity of some of the geometric properties of the arrangement. Namely, it checks that all arrangement vertices are associated with distinct points, and that the halfedges around every vertex are ordered clockwise.

See Also

ArrangementDcel(page 2232)
Arr_default_dcel<Traits> (page 2247)
ArrangementBasicTraits_2(page 2255)
CGAL::is_valid(page 2211)
Insertion functions (page 2212, page 2214, page 2215, page 2216)
Removal functions (page 2219, page 2220)
CGAL::overlay(page 2221)
Input/output functions (page 2228,page 2230)

CGAL::Arr_accessor<Arrangement>

Class

Definition

Arr_accessor<Arrangement> provides an access to some of the private *Arrangement* functions. Users may use these functions to achieve more efficient programs when they have the exact topological information required by the specialized functions.

It is however highly recommended to be very careful when using the accessor functions that modify the arrangement. As we have just mentioned, these functions have very specific requirement on their input on one hand, and perform no preconditions on the other hand, so providing incorrect topological input may invalidate the arrangement.

#include <CGAL/Arr_accessor.h>

Types

Arr_accessor<Arrangement>:: Arrangement_2 — the type of the associated arrangement.

typedef typename Arrangement_2::Point_2

Point_2; — the point type.

typedef typename Arrangement_2::X_monotone_curve_2

X_monotone_curve_2;

the x-monotone curve type.

typedef typename Arrangement_2::Vertex_handle

Vertex_handle;

typedef typename Arrangement_2::Halfedge_handle

Halfedge_handle;

typedef typename Arrangement_2::Face_handle

Face_handle;

typedef typename Arrangement_2::Ccb_halfedge_circulator

Ccb_halfedge_circulator;

represents the boundary of a connected component (CCB).

Creation

Arr_accessor<Arrangement> acc(Arrangement_2& arr);

constructs an accessor attached to the given arrangement *arr*.

Accessing the notification functions

void *acc.notify_before_global_change()*

notifies the arrangement observer that a global change is going to take place (for the usage of the global functions that operate on arrangements).

void *acc.notify_after_global_change()*

notifies the arrangement observer that a global change has taken place (for the usage of the global functions that operate on arrangements).

advanced

Arrangement Predicates

Halfedge_handle *acc.locate_around_vertex(Vertex_handle v, X_monotone_curve_2 c) const*

locates a place for the curve *c* around the vertex *v* and returns a halfedge whose target is *v*, where c should be inserted between this halfedge and the next halfedge around *v* in a clockwise order.

int *acc.halfedge_distance(Halfedge_const_handle e1, Halfedge_const_handle e2) const*

counts the number of edges along the path from *e1* to *e2*. In case the two halfedges do not belong to the same connected component, the function returns (-1).

bool *acc.is_inside_new_face(Halfedge_handle pred1, Halfedge_handle pred2, X_monotone_curve_2 c) const*

determines whether a new halfedge we are about to create, which is to be associated with the curve *c* and directed from *pred1->target()* to *pred2->target()*, lies on the inner CCB of the new face that will be created, introducing this new edge.
Precondition: *pred1->target()* and *pred2->target()* are associated with *c*'s endpoints.
Precondition: *pred1* and *pred2* belong to the same connected component, such that a new face is created by connecting *pred1->target()* and *pred2->target()*.

bool *acc.point_is_in(Point_2 p, Halfedge_const_handle he) const*

determines whether a given point lies within the region bounded by a boundary of the connected component that *he* belongs to. Note that if the function returns *true*, then *p* is contained within *he->face()* (but not on its boundary), or inside one of the holes inside this face, or it may coincide with an isolated vertex in this face.

bool *acc.is_on_outer_boundary(Halfedge_const_handle he) const*

determines whether *he* lies on the outer boundary of its incident face.

bool *acc.is_on_inner_boundary(Halfedge_const_handle he) const*

determines whether *he* lies on the inner boundary of its incident face (that is, whether it lies on the boundary of one of the holes in this face).

Arrangement Modifiers

Vertex_handle *acc.create_vertex(Point_2 p)*

creates a new vertex an associates it with the point *p*. It is the user's responsibility that there is no existing vertex already associated with *p*.

Halfedge_handle *acc.insert_in_face_interior_ex(X_monotone_curve_2 c,*
Face_handle f,
Vertex_handle v1,
Vertex_handle v2,
Comparison_result res)

inserts the curve *c* as a new hole (inner component) of the face *f*, connecting the two isolated vertices *v1* and *v2*. *res* is the comparison result between these two end-vertices. The function returns a handle for one of the new halfedges corresponding to the inserted curve, directed from *v1* to *v2*.
Precondition: It is the user's responsibility that *v1* and *v2* are associated with *c*'s endpoints, that they lie of *f*'s interior and that and that they have no incident edges.

Halfedge_handle *acc.insert_from_vertex_ex(X_monotone_curve_2 c,*
Halfedge_handle pred,
Vertex_handle v,
Comparison_result res)

inserts the curve *c* into the arrangement, such that one of its endpoints corresponds to an arrangement, which is the target vertex of the halfedge *pred*, such that *c* is inserted to the circular list of halfedges around *pred->target()* right between *pred* and its successor. The other end-vertex is given by an isolated vertex *v*, where *res* is the comparison result between *pred->target()* and *v*. The function returns a handle for one of the new halfedges directed from *pred->target()* to *v*.
Precondition: It is the user's responsibility that *pred->target()* and *v* are associated with *c*'s endpoints and that and that *v* has no incident edges.

Halfedge_handle *acc.insert_at_vertices_ex(X_monotone_curve_2 c, Halfedge_handle pred1, Halfedge_handle pred2, Comparison_result res, bool& new_face)*

inserts the curve *c* into the arrangement, such that both *c*'s endpoints correspond to existing arrangement vertices, given by *pred1->target()* and *pred2->target()*. *res* is the comparison result between these two end-vertices. The function creates a new halfedge pair that connects the two vertices (with *pred1* and *pred2* indicate the exact place for these halfedges around the two target vertices) and returns a handle for the halfedge directed from *pred1->target()* to *pred2->target()*. The output flag *new_face* indicates whether a new face has been created following the insertion of the new curve.
Precondition: It is the user's responsibility that *pred1->target()* and *pred2->target()* are associated with *c*'s endpoints and that if a new face is created, then *is_inside_new_face (pred1, pred2, c)* is *true*.

void *acc.insert_isolated_vertex(Face_handle f, Vertex_handle v)*

inserts *v* as an isolated vertex inside *f*.
Precondition: It is the user's responsibility that *v->point()* is contained in the interior of the given face.

void *acc.move_hole(Face_handle f1, Face_handle f2, Ccb_halfedge_circulator hole)*

moves the given hole from the interior of the face *f1* inside the face *f2*.
Precondition: It is the user's responsibility that *hole* is currently contained in *f1* and should be moved to *f2*.

bool *acc.move_isolated_vertex(Face_handle f1, Face_handle f2, Vertex_handle v)*

moves the given isolated vertex from the interior of the face *f1* inside the face *f2*.
Precondition: It is the user's responsibility that *v* is indeed an isolated vertex currently contained in *f1* and should be moved to *f2*.

void *acc.relocate_in_new_face(Halfedge_handle he)*

relocates all holes and isolated vertices to their proper position immediately after a face has split due to the insertion of a new halfedge, namely after *insert_at_vertices_ex()* was invoked and indicated that a new face has been created. *he* is the halfegde returned by *insert_at_vertices_ex()*, such that *he->twin()->face* is the face that has just been split and *he->face()* is the newly created face.

void *acc.relocate_holes_in_new_face(Halfedge_handle he)*

relocates all holes in a new face, as detailed above.

void *acc.relocate_isolated_vertices_in_new_face(Halfedge_handle he)*

relocates all isolated vertices in a new face, as detailed above.

Vertex_handle *acc.modify_vertex_ex(Vertex_handle v, Point_2 p)*

modifies the point associated with the vertex v (the point may be geometrically different than the one currently associated with v). The function returns a handle to the modified vertex (same as v).
Precondition: It is the user's responsibility that no other arrangement vertex is already associated with p.

Halfedge_handle *acc.modify_edge_ex(Halfedge_handle e, X_monotone_curve_2 c)*

modifies the x-monotone curve associated with the edge e (the curve c may be geometrically different than the one currently associated with e). The function returns a handle to the modified edge (same as e).
Precondition: It is the user's responsibility that the interior of c is disjoint from all existing arrangement vertices and edges.

Halfedge_handle *acc.split_edge_ex(Halfedge_handle he,*
Point_2 p,
X_monotone_curve_2 c1,
X_monotone_curve_2 c2)

splits a given edge into two at the split point p, and associate the x-monotone curves *c1* and *c2* with the resulting edges, such that *c1* connects *he->source()* with p and *c2* connects p with *he->target()*. The function return a handle to the split halfedge directed from *he->source()* to the split point p.
Precondition: It is the user's responsibility that the endpoints of *c1* and *c2* correspond to p and to *he*'s end-vertices, as indicated above.

Halfedge_handle *acc.split_edge_ex(Halfedge_handle he,*
Vertex_handle v,
X_monotone_curve_2 c1,
X_monotone_curve_2 c2)

splits a given edge into two at by the vertex v, and associate the x-monotone curves *c1* and *c2* with the resulting edges, such that *c1* connects *he->source()* with v and *c2* connects v with *he->target()*. The function return a handle to the split halfedge directed from *he->source()* to v.
Precondition: It is the user's responsibility that the endpoints of *c1* and *c2* correspond to v and to *he*'s end-vertices, as indicated above. It is also assumed that v has no incident edges.

Face_handle *acc.remove_edge_ex(Halfedge_handle he, bool remove_source = true, bool remove_target = true)*

removes the edge *he* from the arrangement, such that if the edge removal causes the creation of a new hole, *he->target()* lies on the boundary of this hole. The flags *remove_source* and *remove_target* indicate whether the end-vertices of *he* should be removed as well, in case they have no other incident edges. If the operation causes two faces to merge, the merged face is returned. Otherwise, the face to which the edge was incident is returned.

advanced

CGAL::Arrangement_2<Traits,Dcel>::Vertex

Class

Definition

An object *v* of the class *Vertex* represents an arrangement vertex, that is — a 0-dimensional cell, associated with a point on the plane.

Inherits From

typename Dcel::Vertex

Creation

Vertex v; — default constructor.

Access Functions

All non-const methods listed below also have *const* counterparts that return constant handles, iterators or circulators:

bool *v.is_at_open_boundary() const*

checks whether the vertex lies at infinity and not associated with a point with bounded coordinates.

bool *v.is_isolated() const*

checks whether the vertex is isolated (i.e., has no incident edges).

typename Dcel::Size *v.degree() const*

returns the number of edges incident to *v*.

Halfedge_around_vertex_circulator *v.incident_halfedges()*

returns a circulator circulator that allows going over the halfedges incident to *v* (that have *v* as their target). The edges are traversed in a clockwise direction around *v*.
Precondition: *v* is *not* an isolated vertex.

Face_handle *v.face()* — returns a handle to the face that contains *v* in its interior.
Precondition: *v* is an isolated vertex.

typename Traits::Point_2 *v.point() const* — returns the point associated with the vertex.
Precondition: *v* is not a vertex at infinity.

Arr_parameter_space *v.parameter_space_in_x() const*

returns the placement of the *x*-coordinate in the parameter space, that is, either the left boundary-side, the interior, or the right boundary-side.

Arr_parameter_space *v.parameter_space_in_y() const*

returns the placement of the *y*-coordinate in the parameter space, that is, either the bottom boundary-side, the interior, or the top boundary-side.

CGAL::Arrangement_2<Traits,Dcel>::Halfedge

Class

Definition

An object *e* of the class *Halfedge* represents a halfedge in the arrangement. A halfedge is directed from its *source* vertex to its *target* vertex, and has an *incident face* lying to its right. Each halfedge has a *twin* halfedge directed in the opposite direction, where the pair of twin halfedges form together an arrangement edge, that is — a 1-dimensional cell, associated with planar *x*-monotone curve.

Halfedges are stored in doubly-connected lists and form chains. These chains define the inner and outer boundaries of connected components.

Inherits From

typename Dcel::Halfedge

Creation

Halfedge e;	default constructor.

Access Functions

All non-const methods listed below also have *const* counterparts that return constant handles, iterators or circulators:

bool	*e.is_fictitious() const*	
		returns whether the halfedge is fictitious (i.e., connects two vertices at infinity and is not associated with a valid curve).
Vertex_handle	*e.source()*	returns a handle for the source vertex of *e*.
Vertex_handle	*e.target()*	returns a handle for the target vertex of *e*.
Arr_halfedge_direction	*e.direction() const*	
		returns the direction of the halfedge: *ARR_LEFT_TO_RIGHT* if *e*'s source vertex is lexicographically smaller than it target (so the halfedge is directed from left to right), and *ARR_RIGHT_TO_LEFT* if it is lexicographically larger than the target (so the halfedge is directed from right to left).
Face_handle	*e.face()*	returns the face that *e* is incident to (The face lies to the left of *e*).
Halfedge_handle	*e.twin()*	returns the twin halfedge.
Halfedge_handle	*e.prev()*	returns *e*'s predecessor in the connected component it belongs to.

Halfedge_handle	*e.next()*	returns e's successor in the connected component it belongs to.
Ccb_halfedge_circulator	*e.ccb()*	returns a circulator that allows traversing the halfedges of the connected component boundary (CCB) that contains e. The circulator is initialized to point to e.

typename Traits::X_monotone_curve_2

e.curve() const

returns the x-monotone curve associated with e.
Precondition: e is not a fictitious halfedge.

CGAL::Arrangement_2<Traits,Dcel>::Face

Class

Definition

An object of the class *Face* represents an arrangement face, namely, a 2-dimensional arrangement cell. An arrangement that supports only bounded curves contains exactly one *unbounded* face, and a number of bounded faces. An arrangement that supports also unbounded curves has one or more unbounded faces. Such an arrangement has also exactly one fictitious face, which does not correspond to a real two-dimensional cell of the arrangement (and thus it is ignored in counting the number of faces of the arrangement.) Each bounded face has an outer boundary comprising a halfedge chain winding in counterclockwise orientation around it. Each unbounded face of an arrangement that has a fictitious face also has a boundary comprising a counterclockwise halfedge-chain. The edges on the boundary of a face incident to the fictitious face are fictitious, as they do not correspond to real curves. A face may also contain holes, which are defined by clockwise-oriented halfedge chains, and isolated vertices.

Inherits From

typename Dcel::Face

Creation

Face f;	default constructor.

Access Functions

All non-const methods listed below also have *const* counterparts that return constant handles, iterators or circulators:

bool	*f.is_fictitious() const*	returns a Boolean indicating whether this is the fictitious face, which contain the entire arrangement (and does not have an outer CCB). An arrangement that supports only bounded curves does not have a fictitious face at all.
bool	*f.is_unbounded() const*	returns a Boolean indicating whether the face is unbounded.
bool	*f.has_outer_ccb() const*	returns a Boolean indicating whether the face has an outer CCB. (The fictitious face and the unbounded face of an arrangement that does not have a fictitious face do not have outer CCBs.)

Ccb_halfedge_circulator

	f.outer_ccb()	returns a circulator that enables traversing the outer boundary of f. The edges along the CCB are traversed in a counterclockwise direction. *Precondition*: The face f has an outer CCB.
Hole_iterator	*f.holes_begin()*	returns an iterator for traversing all the holes (inner CCBs) of f.
Hole_iterator	*f.holes_end()*	returns a past-the-end iterator for the holes of f.

Isolated_vertex_iterator

f.isolated_vertices_begin()

returns an iterator for traversing all the isolated vertices contained in the interior of f.

Isolated_vertex_iterator

f.isolated_vertices_end()

returns a past-the-end iterator for the isolated vertices contained inside f.

CGAL::is_valid

Definition

The function *is_valid* checks the validity of a given arrangement.

#include <CGAL/Arrangement_2.h>

template<typename Traits, typename Dcel>
bool *is_valid(Arrangement_2<Traits, Dcel> arr)*

Invokes the member function *arr.is_valid()* to verify the topological correctness of the arrangement. Then it performs additional validity tests. It checks that all x-monotone curves associated with arrangement edges are pairwise disjoint in their interior. Then it makes sure that all holes and all isolated vertices are located within the proper arrangement faces. Note that the test carried out by this function may take a considerable amount of time; it is recommended to be used only for debugging purposes.

Requirements

The instantiated traits class must model the concept *ArranagmentXMonotoneTraits_2*.

Function

CGAL::insert

Definition

The function *insert* inserts one or more curves or x-monotone curves into a given arrangement, where no restrictions are imposed on the inserted curves. If an inserted curve is not x-monotone curve, it is subdivided into x-monotone subcurves (and perhaps isolated points), which are inserted into the arrangement.

#include <CGAL/Arrangement_2.h>

template<class Traits, class Dcel, class Curve, class PointLocation>
void *insert(Arrangement_2<Traits,Dcel>& arr, Curve c, PointLocation pl = walk_pl)*

Inserts the given curve *c* into the arrangement *arr*. *c* is subdivided into x-monotone subcurves (and perhaps isolated points). Each subcurve is in turn inserted into the arrangement by locating its left endpoint and computing its zone until reaching the right endpoint.

The given point-location object *pl* is used to locate the left endpoints of the x-monotone curves. By default, the function uses the "walk along line" point-location strategy — namely an instance of the class *Arr_walk_along_line_point_location<Arrangement_2<Traits,Dcel> >*.

Precondition: If provided, *pl* must be attached to the given arrangement *arr*.

template<typename Traits, typename Dcel>
void *insert(Arrangement_2<Traits,Dcel>& arr,*
typename Traits::X_monotone_curve_2 xc,
Object obj)

Inserts the*x-monotone (only)* curve *xc* into the arrangement *arr*. The object *obj*, which either wraps a *Vertex_const_handle*, a *Halfedge_const_handle*, or a *Face_const_handle*, represents the location of *xc*'s left endpoint in the arrangement. The zone of *xc* is computed starting from the feature represented by *obj*. As in the case above, the zone computation terminates, when the right endpoint is reached. Thus, point-location is not required.

template<class Traits, class Dcel, class InputIterator>
void *insert(Arrangement_2<Traits,Dcel>& arr, InputIterator first, InputIterator last)*

Aggregately inserts the curves or x-monotone curves in the range *[first,last)* into the arrangement *arr* using the sweep-line framework.

#include <CGAL/Arrangement_with_history_2.h>

template<typename Traits, typename Dcel, typename PointLocation>
typename Arrangement_with_history_2<Traits,Dcel>::Curve_handle

insert(Arrangement_with_history_2<Traits,Dcel>& arr,
typename Traits::Curve_2 c,
PointLocation pl = walk_pl)

Inserts the given curve *c* into the arrangement with history *arr*, and returns a handle to the inserted curve. *c* is subdivided into x-monotone subcurves (and perhaps isolated points). Each subcurve is in turn inserted into the arrangement by locating its left endpoint and computing its zone until reaching the right endpoint.

The given point-location object *pl* is used to locate the left endpoints of the x-monotone curves. By default, the function uses the "walk along line" point-location strategy — namely an instance of the class *Arr_walk_along_line_point_location<Arrangement_2<Traits,Dcel> >*.

Precondition: If provided, *pl* is attached to the given arrangement *arr*.

template<class Traits, class Dcel, InputIterator>
void *insert(Arrangement_with_history_2<Traits,Dcel>& arr, InputIterator first, InputIterator last)*

Aggregately inserts the curves in the range *[first,last)* into the arrangement with history *arr* using the sweep-line framework.

Requirements

- If the curve is x-monotone curve then The instantiated *Traits* class must model the *ArrangementXMonotoneTraits_2* concept. In case that the curve is not x-monotone then the instantiated *Traits* class must model the *ArrangementTraits_2* concept. That is, it should define the *Curve_2* type, and support its subdivision into x-monotone subcurves (and perhaps isolated points).
- The point-location object *pl*, must model the *ArrangementPointLocation_2* concept.

Function

CGAL::insert_non_intersecting_curve

Definition

The function *insert_non_intersecting_curve* inserts a given x-monotone curve into a given arrangement, where the interior of the given curve is disjoint from all existing arrangement vertices and edges. Under this assumption, it is possible to locate the endpoints of the given curve in the arrangement, and use one of the specialized insertion member-functions of the arrangement according to the results. The insertion operations creates a single new edge, that is, two twin halfedges, and the function returns a handle for the one directed lexicographically in increasing order (from left to right).

A given point-location object is used for answering the two point-location queries on the given curve endpoints. By default, the function uses the "walk along line" point-location strategy — namely, an instance of the class *Arr_walk_along_line_point_location<Arrangement_2<Traits,Dcel> >*.

#include <CGAL/Arrangement_2.h>

```
template<typename Traits, typename Dcel, typename PointLocation>
typename Arrangement_2<Traits,Dcel>::Halfedge_handle

        insert_non_intersecting_curve( Arrangement_2<Traits,Dcel>& arr,
                                       typename Traits::X_monotone_curve_2 xc,
                                       PointLocation pl = walk_pl)
```

Precondition: If provided, *pl* must be attached to the given arrangement *arr*.

Requirements

- The instantiated *Traits* class must model the restricted *ArrangementBasicTraits_2* concept, as no intersections are computed.
- The point-location object *pl* must model the *ArrangementPointLocation_2* concept.

CGAL::insert_non_intersecting_curves

Function

Definition

The function *insert_non_intersecting_curves* inserts a set of x-monotone curves in a given range into a given arrangement. The insertion is performed in an aggregated manner, using the sweep-line algorithm. The input curves should be pairwise disjoint in their interior and pairwise interior-disjoint from all existing arrangement vertices and edges.

#include <CGAL/Arrangement_2.h>

```
template<typename Traits, typename Dcel, InputIterator>
void    insert_non_intersecting_curves( Arrangement_2<Traits,Dcel>& arr,
                                        InputIterator first,
                                        InputIterator last)
```

Requirements

- The instantiated *Traits* class must model the *ArrangementBasicTraits_2* concept, as no intersections are computed.
- *InputIterator::value_type* must be *Traits::X_monotone_curve_2*

Function

CGAL::insert_point

Definition

The function *insert_point* inserts a given point into a given arrangement. It uses a given point-location object to locate the given point in the given arrangement. If the point conincides with an existing vertex, there is nothing left to do; if it lies on an edge, the edge is split at the point. Otherwise, the point is contained inside a face, and is inserted as an isolated vertex inside this face. By default, the function uses the "walk along line" point-location strategy — namely, an instance of the class *Arr_walk_along_line_point_location<Arrangement_2<Traits,Dcel> >*. In either case, the function returns a handle for the vertex associated with the point.

#include <CGAL/Arrangement_2.h>

template<typename Traits, typename Dcel, typename PointLocation>
typename Arrangement_2<Traits,Dcel>::Vertex_handle

insert_point(Arrangement_2<Traits,Dcel>& arr,
typename Traits::Point_2 p,
PointLocation pl = walk_pl)

Precondition: If provided, *pl* must be attached to the given arrangement *arr*.

Requirements

- The instantiated *Traits* class must model the *ArrangementXMonotoneTraits_2* concept. Not all expressions listed by this concept are required. In fact the traits class must model the *ArrangementBasicTraits_2* concept, and support the splitting functionality.
- The point-location object *pl*, must model the *ArrangementPointLocation_2* concept.

CGAL::zone

Function

Definition

The function *zone* compute the zone of the given x-monotone curve in the existing arrangement. Meaning, it output the arrangement's vertices, edges and faces that the x-monotone curve intersects. The order of the objects is the order that they are discovered when traversing the x-monotone curve from left to right.

A given point-location object is used for answering point-location queries during the insertion process. By default, the function uses the "walk along line" point-location strategy — namely an instance of the class *Arr_walk_along_line_point_location<Arrangement_2<Traits,Dcel> >*.

#include <CGAL/Arrangement_2.h>

```
template <class GeomTraits, class TopTraits, class OutputIterator, class PointLocation>
OutputIterator        zone( Arrangement_on_surface_2<GeomTraits, TopTraits>& arr,
                            typename GeomTraits::X_monotone_curve_2 c,
                            OutputIterator oi,
                            PointLocation pl)
```

Compute the zone of the given x-monotone curve c in the arrangement *arr*.

Precondition: If provided, *pl* must be attached to the given arrangement *arr*.

Requirements

- The instantiated *GeomTraits* class must model the *ArrangementXMonotoneTraits_2* concept.
- The point-location object *pl*, must model the *ArrangementPointLocation_2* concept.

Function

CGAL::do_intersect

Definition

The function *do_intersect* checks if a given curve or x-monotone curve intersects an existing arrangement's edges or vertices.

If the give curve is not an x-monotone curve then the function subdivides the given curve into x-monotone subcurves and isolated vertices . Each subcurve is in turn checked for intersection. The function uses the zone algorithm to check if the curve intersects the arrangement. First, the curve's left endpoint is located. Then, its zone is computed starting from its left endpoint location. The zone computation terminates when an intersection with an arrangement's edge/vertex is found or when the right endpoint is reached.

A given point-location object is used for locating the left endpoint of the given curve in the existing arrangement. By default, the function uses the "walk along line" point-location strategy — namely an instance of the class *Arr_walk_along_line_point_location<Arrangement_2<Traits,Dcel> >*.

#include <CGAL/Arrangement_2.h>

```
template <class GeomTraits, class TopTraits, class Curve, class PointLocation>
bool          do_intersect( Arrangement_on_surface_2<GeomTraits, TopTraits>& arr,
                            Curve c,
                            PointLocation pl)
```

Checks if the given curve or x-monotone curve c intersects edges or vertices of the existing arrangement *arr*.

Precondition: If provided, *pl* must be attached to the given arrangement *arr*.

Requirements

- If c is x-monotone then the instantiated *GeomTraits* class must model the *ArrangementXMonotoneTraits_2* concept. If c is a curve then the instantiated *GeomTraits* class must model the *ArrangementTraits_2* concept. That is, it should define the *Curve_2* type, and support its subdivision into x-monotone subcurves (and perhaps isolated points).
- The point-location object *pl*, must model the *ArrangementPointLocation_2* concept.

CGAL::remove_edge

Function

Definition

The function *remove_edge* removes an edge given by one of the twin halfedges that forms it, from a given arrangement. Once the edge is removed, if the vertices associated with its endpoints become isolated, they are removed as well. The call *remove_edge(arr, e)* is equivalent to the call *arr.remove_edge (e, true, true)*. However, this free function requires that *Traits* be a model of the refined concept *ArrangementXMonotoneTraits_2*, which requires merge operations on x-monotone curves. If one of the end-vertices of the given edge becomes redundant after the edge is removed (see *remove_vertex()* for the definition of a redundant vertex), it is removed, and its incident edges are merged. If the edge-removal operation causes two faces to merge, the merged face is returned. Otherwise, the face to which the edge was incident before the removal is returned.

#include <CGAL/Arrangement_2.h>

```
template<typename Traits, typename Dcel>
typename Arrangement_2<Traits,Dcel>::Face_handle

        remove_edge( Arrangement_2<Traits,Dcel>& arr,
                     typename Arrangement_2<Traits,Dcel>::Halfedge_handle e)
```

Requirements

- The instantiated traits class must model the concept *ArrangementXMonotoneTraits_2*.

CGAL::remove_vertex

Definition

The function *remove_vertex* attempts to removed a given vertex from a given arrangement. The vertex can be removed if it is either an isolated vertex, (and has no incident edge,) or if it is a *redundant* vertex. That is, it has exactly two incident edges, whose associated curves can be merged to form a single x-monotone curve. The function returns a boolean value that indicates whether it succeeded removing the vertex from the arrangement.

#include <CGAL/Arrangement_2.h>

```
template <typename Traits, typename Dcel>
bool        remove_vertex( Arrangement_2<Traits,Dcel>& arr,
                           typename Arrangement_2<Traits,Dcel>::Vertex_handle v)
```

Requirements

- The instantiated *Traits* class must model the *ArrangementXMonotoneTraits_2* concept. Not all expressions listed by this concept are required. In fact the traits class must model the *ArrangementBasicTraits_2* concept and support the merging functionality.

CGAL::overlay

Definition

The function *overlay* computes the overlay of two input arrangement objects, and returns the overlaid arrangement. All three arrangements can be instantiated with different geometric traits classes and different DCEL (encapsulated in the various topology-traits classes) classes. The geometry traits of the resulting arrangement is used to construct the resulting arrangement. This means that all the types (e.g., *Traits::Point_2*, *Traits::Curve_2*, and *Traits::Point_2*) of both input arrangements have to be convertible to the types in the resulting arrangement. A given overlay-traits object is used to properly construct the overlaid DCEL that represents the resulting arrangement.

#include <CGAL/Arr_overlay_2.h>

```
template <class GeomTraitsA, class GeomTraitsB, class GeomTraitsRes, class TopTraitsA, class TopTraitsB,
class TopTraitsRes, class OverlayTraits>
void    overlay( Arrangement_2<GeomTraitsA, TopTraitsA> arr1,
                 Arrangement_2<GeomTraitsB, TopTraitsB> arr2,
                 Arrangement_2<GeomTraitsRes, TopTraitsRes>& arr_res,
                 OverlayTraits& ovl_tr)
```

Computes the overlay of two arrangements *arr1* and *arr2*, and sets the output arrangement *res* to represent the overlaid arrangement.

Precondition: *res* does not refer to either *arr1* or *arr2* (that is, “self overlay” is not supported).

```
template <class GeomTraitsA, class GeomTraitsB, class GeomTraitsRes, class TopTraitsA, class TopTraitsB,
class TopTraitsRes>
void    overlay( Arrangement_2<GeomTraitsA, TopTraitsA> arr1,
                 Arrangement_2<GeomTraitsB, TopTraitsB> arr2,
                 Arrangement_2<GeomTraitsRes, TopTraitsRes>& arr_res)
```

Computes the (simple) overlay of two arrangements *arr1* and *arr2*, and sets the output arrangement *res* to represent the overlaid arrangement. It employs the default overlay-traits, which practically does nothing.

Precondition: *res* does not refer to either *arr1* or *arr2* (that is, “self overlay” is not supported).

#include <CGAL/Arrangement_with_history_2.h>

```
template<typename Traits, typename Dcel1, typename Dcel2, typename ResDcel, typename OverlayTraits>
void    overlay( Arrangement_with_history_2<Traits,Dcel1> arr1,
                 Arrangement_with_history_2<Traits,Dcel2> arr2,
                 Arrangement_with_history_2<Traits,ResDcel>& res,
                 OverlayTraits& ovl_tr)
```

Computes the overlay of two arrangements with history *arr1* and *arr2*, and sets the output arrangement with history *res* to represent the overlaid arrangement. The function also constructs a consolidated set of curves that induce *res*.

Precondition: *res* does not refer to either *arr1* or *arr2* (that is, "self overlay" is not supported).

template<typename Traits, typename Dcel1, typename Dcel2, typename ResDcel>
void overlay(Arrangement_with_history_2<Traits,Dcel1> arr1,
Arrangement_with_history_2<Traits,Dcel2> arr2,
Arrangement_with_history_2<Traits,ResDcel>`&` *res)*

Computes the (simple) overlay of two arrangements with history *arr1* and *arr2*, and sets the output arrangement with history *res* to represent the overlaid arrangement. The function also constructs a consolidated set of curves that induce *res*. It employs the default overlay-traits, which practically does nothing.

Precondition: *res* does not refer to either *arr1* or *arr2* (that is, "self overlay" is not supported).

Requirements

- The overlay-traits object *ovl_tr* must model the *OverlayTraits* concept, which is able to construct records of the *ResDcel* class on the basis of the *Dcel1* and *Dcel2* records that induce them.

See Also

OverlayTraits(page 2223)

OverlayTraits

A model for the OverlayTraits should be able to operate on records (namely, vertices, halfedges and faces) of two input DCEL classes, named *Dcel_A* and *Dcel_B*, and construct the records of an output DCEL class, referred to as *Dcel_R*.

Models for the concept are used by the global *overlay()* function to maintain the auxiliary data stored with the DCEL records of the resulting overlaid arrangement, based on the contents of the input records.

Types

OverlayTraits:: Vertex_handle_A	a constant handle a vertex in *Dcel_A*.
OverlayTraits:: Halfedge_handle_A	a constant handle to a halfedge in *Dcel_A*.
OverlayTraits:: Face_handle_A	a constant handle to a face *Dcel_A*.
OverlayTraits:: Vertex_handle_B	a constant handle to a vertex in *Dcel_B*.
OverlayTraits:: Halfedge_handle_B	a constant handle to a halfedge in *Dcel_B*.
OverlayTraits:: Face_handle_B	a constant handle to a face in *Dcel_B*.
OverlayTraits:: Vertex_handle_R	a handle to a vertex in *Dcel_R*.
OverlayTraits:: Halfedge_handle_R	a handle to a halfedge in *Dcel_R*.
OverlayTraits:: Face_handle_R	a handle to a faces in *Dcel_R*.

Functions

Whenever a vertex in the overlaid arrangement is created, one of the following functions is called in order to attach the appropriate auxiliary data to this vertex:

```
void    ovl_tr.create_vertex( Vertex_handle_A v1,
                              Vertex_handle_B v2,
                              Vertex_handle_R v) const
```

constructs the vertex *v* induced by the coinciding vertices *v1* and *v2*.

```
void    ovl_tr.create_vertex( Vertex_handle_A v1,
                              Halfedge_handle_B e2,
                              Vertex_handle_R v) const
```

constructs the vertex *v* induced by the vertex *v1* that lies on the halfedge *e2*.

```
void    ovl_tr.create_vertex( Vertex_handle_A v1,
                              Face_handle_B f2,
                              Vertex_handle_R v) const
```

constructs the vertex *v* induced by the vertex *v1* that lies inside the face *f2*.

void *ovl_tr.create_vertex(Halfedge_handle_A e1, Vertex_handle_B v2, Vertex_handle_R v) const*

constructs the vertex *v* induced by the vertex *v2* that lies on the halfedge *e1*.

void *ovl_tr.create_vertex(Face_handle_A f1, Vertex_handle_B v2, Vertex_handle_R v) const*

constructs the vertex *v* induced by the vertex *v2* that lies inside the face *f1*.

void *ovl_tr.create_vertex(Halfedge_handle_A e1, Halfedge_handle_B e2, Vertex_handle_R v) const*

constructs the vertex *v* induced by the intersection of the halfedges *e1* and *e2*.

Whenever an edge in the overlaid arrangement is created, one of the following functions is called in order to attach the appropriate auxiliary data to this vertex. Note that an edge is created after both its end-vertices are created, (and the corresponding *create_vertex()* methods were invoked). In all cases, the edge is represented by a halfedge *e* directed in lexicographic decreasing order (from right to left). The *create_edge()* method should attach auxiliary data to the twin halfedge (namely to *e->twin()*) as well:

void *ovl_tr.create_edge(Halfedge_handle_A e1, Halfedge_handle_B e2, Halfedge_handle_R e) const*

constructs the halfedge *e* induced by an overlap between the halfedges *e1* and *e2*.

void *ovl_tr.create_edge(Halfedge_handle_A e1, Face_handle_B f2, Halfedge_handle_R e) const*

constructs the halfedge *e* induced by the halfedge *e1* that lies inside the face *f2*.

void *ovl_tr.create_edge(Face_handle_A f1, Halfedge_handle_B e2, Halfedge_handle_R e) const*

constructs the halfedge *e* induced by the halfedge *e2* that lies inside the face *f1*.

The following function is invoked whenever a new face is created. It is guaranteed that all halfedges along the face boundary have already been created an have their auxiliary data fields attached to them:

void	*ovl_tr.create_face(Face_handle_A f1, Face_handle_B f2, Face_handle_R f) const*
	constructs the face *f* induced by the an overlap between the faces *f1* and *f2*.

Has Models

Arr_default_overlay_traits<Arrangement> (page 2226)
Arr_face_overlay_traits<Arr1,Arr2,ResArr,OvlFaceData> (page 2227)

See Also

overlay (page 2221)

Class

CGAL::Arr_default_overlay_traits<Arrangement>

Definition

An instance of *Arr_default_overlay_traits<Arrangement>* should be used for overlaying two arrangements of type *Arrangement* that store no auxiliary data with their DCEL records, where the resulting overlaid arrangement stores no auxiliary DCEL data as well. This class simply gives empty implementation for all traits-class functions.

#include <CGAL/Arr_default_overlay_traits.h>

Is Model for the Concepts

OverlayTraits

See Also

overlay (page 2221)

CGAL::Arr_face_overlay_traits<Arr_A,Arr_B,Arr_R,OvlFaceData>

Class

Definition

An instance of *Arr_face_overlay_traits<Arr_A,Arr_B,Arr_R,OvlFaceData>* should be used for overlaying two arrangements of types *Arr_A* and *Arr_B*, which are instantiated using the same geometric traits-class and with the DCEL classes *Dcel_A* and *Dcel_B* respectively, in order to store their overlay in an arrangement of type *Arr_R*, which is instantiated using a third DCEL class *Dcel_R*. All three DCEL classes are assumed to be instantiations of the *Arr_face_extended_dcel* template with types *FaceData_A*, *FaceData_B* and *FaceData_R*, respectively.

This class gives empty implementation for all overlay traits-class functions, except the function that computes the overlay of two faces. In this case, it uses the functor *OvlFaceData*, which accepts a *FaceData_A* object and a *FaceData_B* object and computes a corresponding *FaceData_R* object, in order to set the auxiliary data of the overlay face.

#include <CGAL/Arr_default_overlay_traits.h>

Is Model for the Concepts

OverlayTraits

See Also

overlay (page 2221)
CGAL::Arr_face_extended_dcel<Traits,FData,V,H,F> (page 2248)

Function

CGAL::read

Definition

The function *read* reads a given arrangement from a given input stream using a specific input format.

#include <CGAL/IO/Arr_iostream.h>

template<typename Traits, typename Dcel, typename Formatter>
std::istream& *read(Arrangement_2<Traits,Dcel>& arr, std::istream& is, Formatter& formatter)*

Reads the arrangement object *arr* from the given input stream *is* using a specific input format defined by formatter.

#include <CGAL/IO/Arr_with_history_iostream.h>

template<typename Traits, typename Dcel, typename WithHistoryFormatter>
std::istream& *read(Arrangement_with_history_2<Traits,Dcel>& arr,*
std::istream& is,
WithHistoryFormatter& formatter)

Reads the arrangement-with-history object *arr* from the given input stream *is* using a specific input format defined by formatter.

Requirements

- The instantiated *Formatter* class must model the *ArrangementInputFormatter* concept.
- The instantiated *WithHistoryFormatter* class must model the *ArrangementWithHistoryInputFormatter* concept.

See Also

write(page 2230)
operator<< page 2231
operator>> page 2229

CGAL::operator>>

Definition

Extracts an arrangement from a given input stream using the input format defined by the *Arr_text_formatter* class — that is, only the basic geometric and topological features of the arrangement are read and no auxiliary data is attached to the DCEL features.

#include <CGAL/IO/Arr_iostream.h>

template<class Traits, class Dcel>
std::istream& *std::istream& is >> Arrangement_2<Traits,Dcel>& arr*

Extracts an arrangement-with-history from a given input stream using the default input format.

#include <CGAL/IO/Arr_with_history_iostream.h>

template<class Traits, class Dcel>
std::istream& *std::istream& is >> Arrangement_with_history_2<Traits,Dcel>& arr*

See Also

read(page 2228)
write(page 2230)
operator>> page 2231

Function

Function

CGAL::write

Definition

The function *write* writes a given arrangement into a given output stream using a specific output format.

#include <CGAL/IO/Arr_iostream.h>

template<typename Traits, typename Dcel, typename Formatter>
std::ostream& *write(Arrangement_2<Traits,Dcel> arr, std::ostream& os, Formatter& formatter)*

Writes the arrangement object *arr* into the given output stream *os* using a specific output format defined by *formatter*.

#include <CGAL/IO/Arr_with_history_iostream.h>

template<typename Traits, typename Dcel, typename WithHistoryFormatter>
std::ostream& *write(Arrangement_with_history_2<Traits,Dcel> arr,*
std::ostream& os,
WithHistoryFormatter& formatter)

Writes the arrangement-with-history object *arr* into the given output stream *os* using a specific output format defined by *formatter*.

Requirements

- The instantiated *Formatter* class must model the *ArrangementOutputFormatter* concept.
- The instantiated *WithHistoryFormatter* class must model the *ArrangementWithHistoryOutputFormatter* concept.

See Also

read(page 2228)
operator<< page 2231
operator>> page 2229

CGAL::operator<<

Definition

The function *operator<<* inserts an object into a given output stream.

#include <CGAL/IO/Arr_iostream.h>

template<typename Traits, typename Dcel>
std::ostream& *std::ostream& os << Arrangement_2<Traits,Dcel> arr*

Inserts the arrangement object *arr* into the output stream *os* using the output format defined by the *Arr_text_formatter* class. Only the basic geometric and topological features of the arrangement are inserted. Auxiliary data that may be attached to the DCEL features is ignored.

#include <CGAL/IO/Arr_with_history_iostream.h>

template<typename Traits, typename Dcel>
std::ostream& *std::ostream& os << Arrangement_with_history_2<Traits,Dcel> arr*

Inserts the arrangement-with-history object *arr* into the output stream *os* using the output format defined by the *Arr_with_history_text_formatter* class. Only the basic geometric and topological features of the arrangement are inserted. Auxiliary data that may be attached to the DCEL features is ignored.

Inserts a polyline into a given *Qt_widget* stream. Only the basic geometric and topological features of the polylines are written. Auxiliary data that might be attached is lost.

#include <CGAL/IO/Qt_widget_Polyline_2.h>

template<typename SegmentTraits>
Qt_widget& *std::ostream& os << Polyline_2<SegmentTraits> polyline*

Inserts a conic arc into a given *Qt_widget* stream. Only the basic geometric and topological features of the conic arcs are written. Auxiliary data that might be attached is lost.

#include <CGAL/IO/Qt_widget_Conic_arc_2.h>

template<typename ConicArc>
Qt_widget& *std::ostream& os << _Conic_x_monotone_arc_2<ConicArc> cv*

See Also

read(page 2228)
write(page 2230)
operator>> page 2229

ArrangementDcel

Definition

A doubly-connected edge-list (DCEL for short) data-structure. It consists of three containers of records: vertices V, halfedges E, and faces F. It maintains the incidence relation among them. The halfedges are ordered in pairs sometimes referred to as twins, such that each halfedge pair represent an edge.

A model of the ArrangementDcel concept must provide the following types and operations. (In addition to the requirements here, the local types *Vertex*,*Halfedge*, *Face Hole* and *Isolated_vertex* must be models of the concepts *ArrangementDcelVertex*(page 2236), *ArrangementDcelHalfedge*(page 2238), *ArrangementDcelFace*(page 2241), *ArrangementDcelHole*(page 2243), and *ArrangementDcelIsolatedVertex* (page 2244) respectively.)

Types

ArrangementDcel:: Vertex	the vertex type.
ArrangementDcel:: Halfedge	the halfedge type.
ArrangementDcel:: Face	the face type.
ArrangementDcel:: Hole	the hole type.
ArrangementDcel:: Isolated_vertex	the isolated vertex type.
ArrangementDcel:: Size	used to represent size values (e.g., *size_t*).
ArrangementDcel:: Vertex_iterator	a bidirectional iterator over the vertices. Its value-type is *Vertex*.
ArrangementDcel:: Halfedge_iterator	a bidirectional iterator over the halfedges. Its value-type is *Halfedge*.
ArrangementDcel:: Face_iterator	a bidirectional iterator over the faces. Its value-type is *Face*.

The non-mutable iterators *Vertex_const_iterator*, *Halfedge_const_iterator* and *Face_const_iterator* are also defined.

Creation

ArrangementDcel dcel; — constructs an empty DCEL with one unbounded face.

*Face** *dcel.assign(Self other, const Face *uf)*

assigns the contents of the *other* DCEL whose unbounded face is given by *uf*, to *dcel*. The function returns a pointer to the unbounded face of *dcel* after the assignment.

Access Functions

Size *dcel.size_of_vertices() const*

returns the number of vertices.

Size	*dcel.size_of_halfedges() const*	
		returns the number of halfedges (always even).
Size	*dcel.size_of_faces() const*	
		returns the number of faces.
Size	*dcel.size_of_holes() const*	
		returns the number of holes (the number of connected components).
Size	*dcel.size_of_isolated_vertices() const*	
		returns the number of isolated vertices.

The following operations have an equivalent *const* operations that return the corresponding non-mutable iterators:

Vertex_iterator	*dcel.vertices_begin()*	returns a begin-iterator of the vertices in *dcel*.
Vertex_iterator	*dcel.vertices_end()*	returns a past-the-end iterator of the vertices in *dcel*.
Halfedge_iterator	*dcel.halfedges_begin()*	
		returns a begin-iterator of the halfedges in *dcel*.
Halfedge_iterator	*dcel.halfedges_end()*	returns a past-the-end iterator of the halfedges in *dcel*.
Face_iterator	*dcel.faces_begin()*	returns a begin-iterator of the faces in *dcel*.
Face_iterator	*dcel.faces_end()*	returns a past-the-end iterator of the faces in *dcel*.

Modifiers

The following operations allocate a new element of the respective type. Halfedges are always allocated in pairs of opposite halfedges. The halfedges and their opposite pointers are automatically set.

*Vertex**	*dcel.new_vertex()*	creates a new vertex.
*Halfedge**	*dcel.new_edge()*	creates a new pair of twin halfedges.
*Face**	*dcel.new_face()*	creates a new face.
*Hole**	*dcel.new_hole()*	creates a new hole record.
*Isolated_vertex**	*dcel.new_isolated_vertex()*	
		creates a new isolated vertex record.

void	*dcel.delete_vertex(Vertex* v)*	
		deletes the vertex v.
void	*dcel.delete_edge(Halfedge* e)*	
		deletes the halfedge e as well as its twin.
void	*dcel.delete_face(Face* f)*	
		deletes the face f.

void *dcel.delete_hole(Hole* ho)*

deletes the hole *ho*.

void *dcel.delete_isolated_vertex(Isolated_vertex* iv)*

deletes the isolated vertex *iv*.

Has Models

Arr_dcel_base<V,H,F>(page 2245)
Arr_default_dcel<Traits> (page 2247)
Arr_face_extended_dcel<Traits,FData,V,H,F> (page 2248)
Arr_extended_dcel<Traits,VData,HData,FData,V,H,F> (page 2249)

See Also

ArrangementDcelVertex(page 2236)
ArrangementDcelHalfedge(page 2238)
ArrangementDcelFace(page 2241)
ArrangementDcelHole(page 2243)
ArrangementDcelIsolatedVertex (page 2244)

ArrangementDcelWithRebind

Definition

The concept ArrangementDcelWithRebind refines the *ArrangementDcel* concept by adding a policy clone idiom in form of a rebind struct-template.

Instantiate a dcel class with many different possible types without ad-hoc limitations on type of the dcel classes.

Refines

ArrangementDcel

Types

ArrangementDcelWithRebind:: template <class T> rebind

allows the instantiation of a model of the base concept *ArrangementDcel* with a different possible geometry-traits class without ad-hoc limitations on it.
Following the standard clone policy, the rebind struct-template must have a nested type named *other* that defines the type of the model replica.

Creation

ArrangementDcelWithRebind dcel;	constructs an empty DCEL with one unbouned face.

Has Models

Arr_default_dcel<Traits> (page 2247)
Arr_face_extended_dcel<Traits,FData,V,H,F> (page 2248)
Arr_extended_dcel<Traits,VData,HData,FData,V,H,F> (page 2249)

Concept

ArrangementDcelVertex

Definition

A vertex record in a DCEL data structure. A vertex is always associated with a point. However, the vertex record only stores a pointer to the associated point, and the actual *Point* object is stored elsewhere.

A vertex usually has several halfedges incident to it, such that it is possible to access one of these halfedges directly and to traverse all incident halfedges around the vertex. However, the DCEL may also contain isolated vertices that have no incident halfedges. In this case, the vertex stores an isolated vertex-information record, indicating the face that contains this vertex in its interior.

Types

ArrangementDcelVertex:: Halfedge	the corresponding DCEL halfedge type.
ArrangementDcelVertex:: Isolated_vertex	the corresponding DCEL isolated vertex-information type.
ArrangementDcelVertex:: Point	the point type associated with the vertex.

Creation

ArrangementDcelVertex *v*;		default constructor.
void	*v.assign(Self other)*	assigns *v* with the contents of the *other* vertex.

Access Functions

All functions below also have *const* counterparts, returning non-mutable pointers or references:

bool	*v.is_isolated() const*	returns whether the vertex is isolated (has no incident halfedges).
*Halfedge**	*v.halfedge()*	returns an incident halfedge that has *v* as its target. *Precondition*: *v* is *not* an isolated vertex.
*Isolated_vertex**	*v.isolated_vertex()*	returns the isolated vertex-information record. *Precondition*: *v* is an isolated vertex.
bool	*v.has_null_point() const*	returns whether the vertex is not associated with a valid point (i.e. it lies at infinity).
Point&	*v.point()*	returns the associated point. *Precondition*: *v*() is associated with a valid point.

Arr_parameter_space

v.parameter_space_in_x() const

returns the placement of the x-coordinate in the parameter space, that is, either the left boundary-side, the interior, or the right boundary-side.

Arr_parameter_space

v.parameter_space_in_y() const

returns the placement of the y-coordinate in the parameter space, that is, either the bottom boundary-side, the interior, or the top boundary-side.

Modifiers

void *v.set_halfedge(Halfedge* e)*

sets the incident halfedge, marking v as a regular vertex.

void *v.set_isolated_vertex(Isolated_vertex* iv)*

sets the isolated vertex-information record, marking v as an isolated vertex.

void *v.set_point(Point* p)* sets the associated point.

void *v.set_boundary(Arr_parameter_space inf_x, Arr_parameter_space inf_y)*

sets v as a vertex on a boundary side.
Precondition: Either *inf_x* or *inf_y* is not *ARR_INTERIOR*.

See Also

ArrangementDcel (page 2232)
ArrangementDcelHalfedge (page 2238)
ArrangementDcelIsolatedVertex (page 2244)

Concept

ArrangementDcelHalfedge

Definition

A halfedge record in a DCEL data structure. Two halfedges with opposite directions always form an edge (a halfedge pair). The halfedges form together chains, defining the boundaries of connected components, such that all halfedges along a chain have the same incident face. Note that the chain the halfedge belongs to may form the outer boundary of a bounded face (an outer CCB) or the boundary of a hole inside a face (an inner CCB).

An edge is always associated with a curve, but the halfedge records only store a pointer to the associated curve, and the actual curve objects are stored elsewhere. Two opposite halfedges are always associated with the same curve.

Types

ArrangementDcelHalfedge:: Vertex	the corresponding DCEL vertex type.
ArrangementDcelHalfedge:: Face	the corresponding DCEL face type.
ArrangementDcelHalfedge:: Hole	the corresponding DCEL hole type.

ArrangementDcelHalfedge:: X_monotone_curve

the curve type associated with the edge.

Creation

ArrangementDcelHalfedge e;	default constructor.

void	*e.assign(Self other)*	assigns *e* with the contents of the *other* halfedge.

Access Functions

Arr_halfedge_direction

e.direction() const	returns *ARR_LEFT_TO_RIGHT* if *e*'s source vertex is lexicographically smaller than it target, and *ARR_RIGHT_TO_LEFT* if it is lexicographically larger than the target.

bool	*e.is_on_hole() const*	determines whether the *e* lies on an outer CCB of a bounded face, or on an inner CCB (a hole inside a face). The function returns *true* if *e* lies on a hole.

All functions below also have *const* counterparts, returning non-mutable pointers or references:

*Halfedge**	*e.opposite()*	returns the twin halfedge.
*Halfedge**	*e.prev()*	returns the previous halfedge along the chain.

*Halfedge**	*e.next()*	returns the next halfedge along the chain.
*Vertex**	*e.vertex()*	returns the target vertex.
*Face**	*e.face()*	returns the incident face. *Precondition*: *e* lies on the outer boundary of this face.
*Hole**	*e.hole()*	returns the hole (inner CCB) *e* belongs to. *Precondition*: *e* lies on a hole inside its incident face.
bool	*e.has_null_curve() const*	returns whether the vertex is not associated with a valid curve.
X_monotone_curve&	*e.curve()*	returns the associated curve. *Precondition*: *e* is associated with a valid curve.

Modifiers

void	*e.set_opposite(Halfedge* opp)*	sets the opposite halfedge.
void	*e.set_direction(Arr_halfedge_direction dir)*	sets the lexicographical order between *e*'s source and target vertices to be *dir*. The direction of the opposite halfedge is also set to the opposite direction.
void	*e.set_prev(Halfedge* prev)*	sets the previous halfedge of *e* along the chain, and updates the cross-pointer *prev->next()*.
void	*e.set_next(Halfedge* next)*	sets the next halfedge of *e* along the chain, and updates the cross-pointer *next->prev()*.
void	*e.set_vertex(Vertex* v)*	sets the target vertex.
void	*e.set_face(Face* f)*	sets the incident face, marking that *e* lies on the outer CCB of the face *f*.

void	*e.set_hole(Hole* ho)*	sets the incident hole, marking that *e* lies on an inner CCB.
void	*e.set_curve(X_monotone_curve* c)*	
		sets the associated curve of *e* and its opposite halfedge.

See Also

ArrangementDcel (page 2232)
ArrangementDcelVertex (page 2236)
ArrangementDcelFace (page 2241)
ArrangementDcelHole (page 2243)

ArrangementDcelFace

Definition

A face record in a DCEL data structure. A face may either be unbounded, otherwise it has an incident halfedge along the chain defining its outer boundary. A face may also contain holes and isolated vertices in its interior.

Types

ArrangementDcelFace:: Vertex	the corresponding DCEL vertex type.
ArrangementDcelFace:: Halfedge	the corresponding DCEL halfedge type.
ArrangementDcelFace:: Hole_iterator	a bidirectional iterator over the holes in inside the face. Its value-type is *Halfedge**.
ArrangementDcelFace:: Isolated_vertex_iterator	a bidirectional iterator over the isolated vertices in inside the face. Its value-type is *Vertex**.

The non-mutable iterators *Hole_const_iterator*, and *Isolated_vertex_const_iterator* are also defined.

Creation

ArrangementDcelFace f;		default constructor.
void	f*.assign(Self other)*	assigns f with the contents of the *other* face.

Access Functions

All functions below also have *const* counterparts, returning non-mutable pointers or iterators:

bool	f*.is_unbounded() const*	returns whether the face is unbounded.
*Halfedge**	f*.halfedge()*	returns an incident halfedge along the outer boundary of the face. If f has no outer boundary, the function returns *NULL*.
size_t	f*.number_of_holes() const*	returns the number of holes inside f.
Hole_iterator	f*.holes_begin()*	returns a begin-iterator for the holes inside f.
Hole_iterator	f*.holes_end()*	returns a past-the-end iterator for the holes inside f.

size_t	*f.number_of_isolated_vertices() const*	
		returns the number of isolated vertices inside f.
Isolated_vertex_iterator		
	f.isolated_vertices_begin()	
		returns a begin-iterator for the isolated vertices inside f.
Isolated_vertex_iterator		
	f.isolated_vertices_end()	
		returns a past-the-end iterator for the isolated vertices inside f.

Modifiers

void	*f.set_unbounded(bool flag)*	
		sets the face as unbounded (if *flag* is *true*), or as a bounded face (if it is *false*).
void	*f.set_halfedge(Halfedge* e)*	
		sets the incident halfedge.
void	*f.add_hole(Halfedge* e)*	
		adds e as a hole inside f.
void	*f.erase_hole(Hole_iterator it)*	
		removes the hole that *it* points to from inside f.
void	*f.add_isolated_vertex(Vertex* v)*	
		adds v as an isolated vertex inside f.
void	*f.erase_isolated_vertex(Isolated_vertex_iterator it)*	
		removes the isolated vertex that *it* points to from inside f.

See Also

ArrangementDcel (page 2232)
ArrangementDcelVertex (page 2236)
ArrangementDcelHalfedge (page 2238)

ArrangementDcelHole

Concept

Definition

A hole record in a DCEL data structure, which stores the face that contains the hole in its interior, along with an iterator for the hole in the holes' container of this face.

Types

ArrangementDcelHole:: Face	the corresponding DCEL face type.

typedef Face::Hole_iterator

Hole_iterator;

Creation

ArrangementDcelHole ho;	default constructor.

Access Functions

All functions below also have *const* counterparts, returning non-mutable pointers or iterators:

*Face**	*ho.face()*	returns the incident face, which contains *ho* in its interior.
Hole_iterator	*ho.iterator()*	returns an iterator for the hole.

Modifiers

void	*ho.set_face(Face* f)*	sets the incident face.
void	*ho.set_iterator(Hole_iterator it)*	sets the hole iterator.

See Also

ArrangementDcel (page 2232)
ArrangementDcelFace (page 2241)

Concept

ArrangementDcelIsolatedVertex

Definition

An isolated vertex-information record in a DCEL data structure, which stores the face that contains the isolated vertex in its interior, along with an iterator for the isolated vertex in the isolated vertices' container of this face.

Types

ArrangementDcelIsolatedVertex:: Face	the corresponding DCEL face type.

typedef Face::Isolated_vertex_iterator

Isolated_vertex_iterator;

Creation

ArrangementDcelIsolatedVertex iv;	default constructor.

Access Functions

All functions below also have *const* counterparts, returning non-mutable pointers or iterators:

*Face**	*iv.face()*	returns the incident face, which contains *iv* in its interior.
Isolated_vertex_iterator	*iv.iterator()*	returns an iterator for the isolated vertex.

Modifiers

void	*iv.set_face(Face* f)*	sets the incident face.
void	*iv.set_iterator(Isolated_vertex_iterator it)*	sets the isolated vertex iterator.

See Also

ArrangementDcel (page 2232)
ArrangementDcelFace (page 2241)

CGAL::Arr_dcel_base<V,H,F>

Definition

The *Arr_dcel_base<V,H,F>* class is an important ingredient in the definition of DCEL data structures. It serves as a basis class for any instance of the *Dcel* template parameter of the *Arrangement_2* template. In particular it is the basis class of the default *Dcel* template parameter, and the basis class of any extended DCEL. The template parameters V, H, and F must be instantiated with models of the concepts *ArrangementDcelVertex*, *ArrangementDcelHalfedge*, and *ArrangementDcelFace* respectively.

#include <CGAL/Arr_dcel_base.h>

Is Model for the Concepts

ArrangementDcel

Class Arr_vertex_base<Point>

Definition

The basic DCEL vertex type. Serves as a basis class for an extended vertex record with auxiliary data fields. The *Point* parameter is the type of points associated with the vertices.

Is Model for the Concepts

ArrangementDcelVertex

Class Arr_halfedge_base<Curve>

Definition

The basic DCEL halfedge type. Serves as a basis class for an extended halfedge record with auxiliary data fields. The *Curve* parameter is the type of x-monotone curves associated with the vertices.

Is Model for the Concepts

ArrangementDcelHalfedge

Class Arr_face_base

Definition

The basic DCEL face type. Serves as a basis class for an extended face record with auxiliary data fields.

Is Model for the Concepts

ArrangementDcelFace

CGAL::Arr_default_dcel<Traits>

Definition

The default DCEL class used by the *Arrangement_2* class-template is parameterized by a traits class, which is a model of the *ArrangementBasicTraits_2* concept. It simply uses the nested *Traits::Point_2* and *Traits::X_monotone_curve_2* to instantiate the base vertex and halfedge types, respectively. Thus, the default DCEL records store no other information, except for the topological incidence relations and the geometric data attached to vertices and edges.

#include <CGAL/Arr_default_dcel.h>

Is Model for the Concepts

ArrangementDcelWithRebind

Inherits From

Arr_dcel_base<Arr_vertex_base<typename Traits_::Point_2>,
Arr_halfedge_base<typename Traits_::X_monotone_curve_2>,
Arr_face_base>

See Also

Arr_dcel_base<V,H,F> (page 2245)

CGAL::Arr_face_extended_dcel<Traits,FData,V,H,F>

Definition

The *Arr_face_extended_dcel<Traits,FData,V,H,F>* class-template extends the DCEL face-records, making it possible to store extra (non-geometric) data with the arrangement faces. The class should be instantiated by an *FData* type which represents the extra data stored with each face.

Note that all types of DCEL features (namely vertex, halfedge and face) are provided as template parameters. However, by default they are defined as follows:

V = Arr_vertex_base<typename Traits::Point_2>
H = Arr_halfedge_base<typename Traits::X_monotone_curve_2>
F = Arr_face_base

#include <CGAL/Arr_extended_dcel.h>

Is Model for the Concepts

ArrangementDcelWithRebind

Inherits From

Arr_dcel_base<V, H, Arr_extended_face<F, FData> >

See Also

Arr_dcel_base<V,H,F> (page 2245)

CGAL::Arr_extended_dcel<Traits,VData,HData,FData,V,H,F>

Class

Definition

The *Arr_extended_dcel<Traits,VData,HData,FData,V,H,F>* class-template extends the topological-features of the DCEL namely the vertex, halfedge, and face types. While it is possible to maintain extra (non-geometric) data with the curves or points of the arrangement by extending their types respectively, it is also possible to extend the vertex, halfedge, or face types of the DCEL through inheritance. As the technique to extend these types is somewhat cumbersome and difficult for inexperienced users, the *Arr_extended_dcel< Traits,VData,HData,FData,V,H,F>* class-template provides a convenient way to do that. Each one of the three features is extended with a corresponding data type provided as parameters. This class template is also parameterized with a traits class used to extract default values for the vertex, halfedge, and face base classes, which are the remaining three template parameters respectively. The default values follow:

V = Arr_vertex_base<typename Traits::Point_2>
H = Arr_halfedge_base<typename Traits::X_monotone_curve_2>
F = Arr_face_base

#include <CGAL/Arr_extended_dcel.h>

Is Model for the Concepts

ArrangementDcelWithRebind

Inherits From

Arr_dcel_base<Arr_extended_vertex<V, VData>,
Arr_extended_halfedge<H, HData>,
Arr_extended_face<F, FData> >

See Also

Arr_dcel_base<V,H,F> (page 2245)

Class

CGAL::Arr_extended_vertex<VertexBase,VData>

Definition

The *Arr_extended_vertex<VertexBase,VData>* class-template extends the vertex topological-features of the DCEL. It is parameterized by a vertex base-type *VertexBase* and a data type *VData* used to extend the vertex base-type.

#include <CGAL/Arr_extended_dcel.h>

Is Model for the Concepts

ArrangementDcelVertex

Inherits From

VertexBase

Creation

void	*v.assign(Self other)*	assigns *v* with the contents of the *other* vertex.

Access Functions

VData	*v.data() const*	obtains the auxiliary data (a non-const version, returning a reference to a mutable data object is also available).

Modifiers

void	*v.set_data(VData data)*	sets the auxiliary data.

See Also

Arr_dcel_base<V,H,F> (page 2245)

CGAL::Arr_extended_halfedge<HalfedgeBase,HData>

Class

Definition

The *Arr_extended_halfedge<HalfedgeBase,HData>* class-template extends the halfedge topological-features of the DCEL. It is parameterized by a halfedge base-type *HalfedgeBase* and a data type *HData* used to extend the halfedge base-type.

#include <CGAL/Arr_extended_dcel.h>

Is Model for the Concepts

ArrangementDcelHalfedge

Inherits From

HalfedgeBase

Creation

void	*he.assign(Self other)*	assigns *he* with the contents of the *other* vertex.

Access Functions

HData	*he.data() const*	obtains the auxiliary data (a non-const version, returning a reference to a mutable data object is also available).

Modifiers

void	*he.set_data(HData data)*	sets the auxiliary data.

See Also

Arr_dcel_base<V,H,F> (page 2245)

Class

CGAL::Arr_extended_face<FaceBase,FData>

Definition

The *Arr_extended_face<FaceBase,FData>* class-template extends the face topological-features of the DCEL. It is parameterized by a face base-type *FaceBase* and a data type *FData* used to extend the face base-type.

#include <CGAL/Arr_extended_dcel.h>

Is Model for the Concepts

ArrangementDcelFace

Inherits From

FaceBase

Creation

void	*f.assign(Self other)*	assigns *f* with the contents of the *other* vertex.

Access Functions

FData	*f.data() const*	obtains the auxiliary data (a non-const version, returning a reference to a mutable data object is also available).

Modifiers

void	*f.set_data(FData data)*	sets the auxiliary data.

See Also

Arr_dcel_base<V,H,F> (page 2245)

CGAL::Arr_curve_end

Definition

The enumeration *Arr_curve_end* is used to indicate one of the two ends of an x-monotone curve. It is used by models of the *ArrangementOpenBoundaryTraits_2* concept.

#include <CGAL/Arr_enums.h>

enum Arr_curve_end { *ARR_MIN_END, ARR_MAX_END*}*;*

See Also

ArrangementOpenBoundaryTraits_2

Enum

CGAL::Arr_halfedge_direction

Definition

The enumeration *Arr_halfedge_direction* is defined by *CGAL::Arrangement_2<Traits,Dcel>::Halfedge* to specify the direction of the halfedge.

#include <CGAL/Arr_enums.h>

enum Arr_halfedge_direction { *ARR_LEFT_TO_RIGHT, ARR_RIGHT_TO_LEFT*}*;*

See Also

CGAL::Arrangement_2<Traits,Dcel>::Halfedge

ArrangementBasicTraits_2

Definition

The concept ArrangementBasicTraits_2 defines the minimal set of geometric predicates needed for the construction and maintenance of objects of the class *Arrangement_2*, as well as performing simple queries (such as point-location queries) on such arrangements.

A model of this concept must define nested *Point_2* and *X_monotone_curve_2* types, which represent planar points and continuous *x*-monotone curves (a vertical segment is also considered to be *weakly* *x*-monotone), respectively. The *x*-monotone curves are assumed to be pairwise disjoint in their interiors, so they do not intersect except at their endpoints.

The *X_monotone_curve_2* curves of an arrangement are confined to an iso-rectangular area called the parameter space. The iso-rectangule can be unbounded, open, or closed. The set of predicates provided by a model the concept ArrangementBasicTraits_2 is sufficient for constructing arrangements of *x*-monotone curves that do not reach or approach the boundary of the parameter space. The nature of the input curves, whether they are expected to reach or approach the left, right, bottom, or top side of the boundary of the parameter space, are conveyed through the definition of four additional nested types, namely *Left_side_category*, *Right_side_category*, *Bottom_side_category*, and *Top_side_category*. Each such type must be convertible to the type *Arr_oblivious_side_tag*.

Refines

DefaultConstructible, CopyConstructible, Assignable

Types

ArrangementBasicTraits_2:: Point_2 — models the concept *ArrTraits::Point_2*.

ArrangementBasicTraits_2:: X_monotone_curve_2 — models the concept *ArrTraits::XMonotoneCurve_2*.

Categories

ArrangementBasicTraits_2:: Has_left_category — indicates whether the nested functor *Compare_at_x_left_2* is provided.

ArrangementBasicTraits_2:: Left_side_category — Must be convertible to *Arr_oblivious_side_tag*.

ArrangementBasicTraits_2:: Bottom_side_category — Must be convertible to *Arr_oblivious_side_tag*.

ArrangementBasicTraits_2:: Top_side_category

Must be convertible to *Arr_oblivious_side_tag*.

ArrangementBasicTraits_2:: Right_side_category

Must be convertible to *Arr_oblivious_side_tag*.

Functor Types

ArrangementBasicTraits_2:: Compare_x_2

models the concept *ArrTraits::CompareX_2*.

ArrangementBasicTraits_2:: Compare_xy_2

models the concept *ArrTraits::CompareXy_2*.

ArrangementBasicTraits_2:: Construct_min_vertex_2

models the concept *ArrTraits::ConstructMinVertex_2*.

ArrangementBasicTraits_2:: Construct_max_vertex_2

models the concept *ArrTraits::ConstructMaxVertex_2*.

ArrangementBasicTraits_2:: Is_vertical_2

models the concept *ArrTraits::IsVertical_2*.

ArrangementBasicTraits_2:: Compare_y_at_x_2

models the concept *ArrTraits::CompareYAtX_2*.

ArrangementBasicTraits_2:: Compare_y_at_x_left_2

models the concept *ArrTraits::CompareYAtXLeft_2*. Required only if the *Has_left_category* category is convertible to *Tag_true*.

ArrangementBasicTraits_2:: Compare_y_at_x_right_2

models the concept *ArrTraits::CompareYAtXRight_2*.

ArrangementBasicTraits_2:: Equal_2

models the concept *ArrTraits::Equal_2*.

Accessing Functor Objects

Compare_x_2	*traits.compare_x_2_object() const*
Compare_xy_2	*traits.compare_xy_2_object() const*
Construct_min_vertex_2	*traits.construct_min_vertex_2_object() const*
Construct_max_vertex_2	*traits.construct_max_vertex_2_object() const*
Is_vertical_2	*traits.is_vertical_2_object() const*
Compare_y_at_x_2	*traits.compare_y_at_x_2_object() const*
Compare_y_at_x_left_2	*traits.compare_y_at_x_left_2_object() const*

Compare_y_at_x_right_2	*traits.compare_y_at_x_right_2_object() const*
Equal_2	*traits.equal_2_object() const*

Has Models

CGAL::Arr_segment_traits_2<Kernel>
CGAL::Arr_non_caching_segment_basic_traits_2<Kernel>
CGAL::Arr_non_caching_segment_traits_2<Kernel>
CGAL::Arr_linear_traits_2<Kernel>
CGAL::Arr_polyline_traits_2<SegmentTraits>
CGAL::Arr_circle_segment_traits_2<Kernel>
CGAL::Arr_line_arc_traits_2<CircularKernel>
CGAL::Arr_circular_arc_traits_2<CircularKernel>
CGAL::Arr_circular_line_arc_traits_2<CircularKernel>
CGAL::Arr_conic_traits_2<RatKernel,AlgKernel,NtTraits>
CGAL::Arr_rational_function_traits_2<AlgebraicKernel_d_1>
CGAL::Arr_Bezier_curve_traits_2<RatKernel,AlgKernel,NtTraits>
CGAL::Arr_algebraic_segment_traits_2<Coefficient>
CGAL::Arr_curve_data_traits_2<Tr,XData,Mrg,CData,Cnv>
CGAL::Arr_consolidated_curve_data_traits_2<Traits,Data>

ArrTraits::Point_2

Definition

Represents a point in the plane.

Refines

DefaultConstructible, CopyConstructible, Assignable

Has Models

ArrangementBasicTraits_2::Point_2

ArrTraits::XMonotoneCurve_2

Concept

Definition

Represents a planar (weakly) x-monotone curve.

Refines

DefaultConstructible, CopyConstructible, Assignable

Has Models

ArrangementBasicTraits_2::X_monotone_curve_2

Concept

ArrTraits::CompareX_2

Refines

AdaptableBinaryFunction

Has Models

ArrangementBasicTraits_2::Compare_x_2

Comparison_result *fo(ArrTraits::Point_2 p1, ArrTraits::Point_2 p2)*

returns *SMALLER*, *EQUAL*, or *LARGER* according to the x-ordering of points *p1* and *p2*.

ArrTraits::CompareXy_2

Refines

AdaptableBinaryFunction

Has Models

ArrangementBasicTraits_2::Compare_xy_2

Comparison_result *fo(ArrTraits::Point_2 p1, ArrTraits::Point_2 p2)*

returns *SMALLER*, *EQUAL*, or *LARGER* according to the lexicographic *xy*-order of the points *p1* and *p2*.

Concept

ArrTraits::ConstructMinVertex_2

Refines

AdaptableUnaryFunction

Has Models

ArrangementBasicTraits_2::Construct_min_vertex_2

ArrTraits::Point_2	*fo(ArrTraits::X_monotone_curve_2 xc)*
	returns the lexicographically smaller (left) endpoint of *xc*.

ArrTraits::ConstructMaxVertex_2

Refines

AdaptableUnaryFunction

Has Models

ArrangementBasicTraits_2::Construct_max_vertex_2

ArrTraits::Point_2 *fo(ArrTraits::X_monotone_curve_2 xc)*

returns the lexicographically larger (right) endpoint of *xc*.

Concept

ArrTraits::IsVertical_2

Refines

AdaptableUnaryFunction

Has Models

ArrangementBasicTraits_2::Is_vertical_2

bool	*fo(ArrTraits::X_monotone_curve_2 xc)*
	determines whether *xc* is a vertical segment.

ArrTraits::CompareYAtX_2

Concept

Refines

AdaptableBinaryFunction

Has Models

ArrangementBasicTraits_2::Compare_y_at_x_2

Comparison_result	*fo(ArrTraits::Point_2 p, ArrTraits::X_monotone_curve_2 xc)*
	compares the y-coordinates of p and the vertical projection of p on xc, and returns *SMALLER*, *EQUAL*, or *LARGER* according to the result.

Concept

ArrTraits::CompareYAtXLeft_2

Refines

AdaptableTernaryFunction

Has Models

ArrangementBasicTraits_2::Compare_y_at_x_left_2

Comparison_result *fo.operator()(ArrTraits::X_monotone_curve_2 xc1, ArrTraits::X_monotone_curve_2 xc2, ArrTraits::Point_2 p)*

accepts two x-monotone curves *xc1* and *xc2* that have a common right endpoint p, and returns *SMALLER*, *EQUAL* or *LARGER* according to the relative position of the two curves immediately to the left of p. Note that in case one of the x-monotone curves is a vertical segment (emanating downward from p), it is always considered to be *below* the other curve.

ArrTraits::CompareYAtXRight_2

Refines

AdaptableTernaryFunction

Has Models

ArrangementBasicTraits_2::Compare_y_at_x_right_2

Comparison_result *fo.operator()(ArrTraits::X_monotone_curve_2 xc1, ArrTraits::X_monotone_curve_2 xc2, ArrTraits::Point_2 p)*

accepts two x-monotone curves *xc1* and *xc2* that have a common left endpoint p, and returns *SMALLER, EQUAL* or *LARGER* according to the relative position of the two curves immediately to the right of p. Note that in case one of the x-monotone curves is a vertical segment emanating upward from p, it is always considered to be *above* the other curve.

Concept

Concept

ArrTraits::Equal_2

Refines

AdaptableBinaryFunction

Has Models

ArrangementBasicTraits_2::Equal_2

bool	*fo(ArrTraits::Point_2 p1, ArrTraits::Point_2 p2)*	
		determines whether *p1* and *p2* are geometrically equivalent.
bool	*fo(ArrTraits::X_monotone_curve_2 xc1, ArrTraits::X_monotone_curve_2 xc2)*	
		determines whether *xc1* and *xc2* are geometrically equivalent (have the same graph).

ArrangementLandmarkTraits_2

Concept

Definition

The concept ArrangementLandmarkTraits_2 refines the general traits concept by adding operations needed for the landmarks point-location strategy, namely — approximating points and connecting points with a simple x-monotone curve.

A model of this concept must define the *Approximate_number_type*, which is used to approximate the coordinates of *Point_2* instances. It is recommended to define the approximated number type as the built-in *double* type.

Refines

ArrangementTraits_2

Types

ArrangementLandmarkTraits_2:: Approximate_number_type

the number type used to approximate point coordinates.

Functor Types

ArrangementLandmarkTraits_2:: Approximate_2

models the concept *ArrTraits::Approximate_2*.

ArrangementLandmarkTraits_2:: Construct_x_monotone_curve_2

models the concept *ArrTraits::ConstructXMonotoneCurve_2*.

Accessing Functor Objects

Approximate_2	*traits.approximate_2_object() const*
Construct_x_monotone_curve_2	*traits.construct_x_monotone_curve_2_object() const*

Has Models

CGAL::Arr_non_caching_segment_traits_2<Kernel>
CGAL::Arr_segment_traits_2<Kernel>
CGAL::Arr_linear_traits_2<Kernel>
CGAL::Arr_polyline_traits_2<SegmentTraits>
CGAL::Arr_conic_traits_2<RatKernel,AlgKernel,NtTraits>

See Also

ArrangementTraits_2(page 2279)

ArrTraits::Approximate_2

Concept

Refines

Functor

Has Models

ArrangementLandmarkTraits_2::Approximate_2

Approximate_number_type

fo(ArrTraits::Point_2 p, int i)

returns an approximation of p's x-coordinate (if i == *0*), or of p's y-coordinate (if i == *1*).

Concept

ArrTraits::ConstructXMonotoneCurve_2

Refines

Functor

Has Models

ArrangementLandmarkTraits_2::Construct_x_monotone_curve_2

ArrTraits::X_monotone_curve_2

fo(ArrTraits::Point_2 p1, ArrTraits::Point_2 p2)

returns an x-monotone curve connecting *p1* and *p2* (i.e., the two input points are its endpoints).

ArrangementXMonotoneTraits_2

Concept

Definition

The concept ArrangementXMonotoneTraits_2 refines the basic arrangement-traits concept. A model of this concept is able to handle x-monotone curves that intersect in their interior (and points that coincide with curve interiors). This is necessary for constructing arrangements of sets of intersecting x-monotone curves.

As the resulting structure, represented by the *Arrangement_2* class, stores pairwise interior-disjoint curves, the input curves are split at the intersection points before being inserted into the arrangement. A model of this refined concept therefore needs to compute the intersections (and possibly overlaps) between two x-monotone curves and to support curve splitting.

Refines

ArrangementBasicTraits_2

Types

ArrangementXMonotoneTraits_2:: Multiplicity	the multiplicity type.

Tags

ArrangementXMonotoneTraits_2:: Has_merge_category	indicates whether the nested functors *Are_mergeable_2* and *Merge_2* are provided.

Functor Types

ArrangementXMonotoneTraits_2:: Intersect_2

models the concept *ArrTraits::Intersect_2*.

ArrangementXMonotoneTraits_2:: Split_2

models the concept *ArrTraits::Split_2*.

The two following function-object types are optional. If they are supported, the *Has_merge_category* tag should be defined as *Tag_true* (and *Tag_false* otherwise.

ArrangementXMonotoneTraits_2:: Are_mergeable_2

models the concept *ArrTraits::AreMergeable_2*.

ArrangementXMonotoneTraits_2:: Merge_2

models the concept *ArrTraits::Merge_2*.

Accessing Functor Objects

Intersect_2	*traits.intersect_2_object() const*
Split_2	*traits.split_2_object() const*

The two following methods are optional. If they are supported, the *Has_merge_category* tag should be defined as *Tag_true* (and *Tag_false* otherwise.

Are_mergeable_2	*traits.are_mergeable_2_object() const*
Merge_2	*traits.merge_2_object() const*

Has Models

CGAL::Arr_segment_traits_2<Kernel>
CGAL::Arr_non_caching_segment_traits_2<Kernel>
CGAL::Arr_linear_traits_2<Kernel>
CGAL::Arr_polyline_traits_2<SegmentTraits>
CGAL::Arr_circle_segment_traits_2<Kernel>
CGAL::Arr_line_arc_traits_2<CircularKernel>
CGAL::Arr_circular_arc_traits_2<CircularKernel>
CGAL::Arr_circular_line_arc_traits_2<CircularKernel>
CGAL::Arr_conic_traits_2<RatKernel,AlgKernel,NtTraits>
CGAL::Arr_rational_function_traits_2<AlgebraicKernel_d_1>
CGAL::Arr_Bezier_curve_traits_2<RatKernel,AlgKernel,NtTraits>
CGAL::Arr_algebraic_segment_traits_2<Coefficient>
CGAL::Arr_curve_data_traits_2<Tr,XData,Mrg,CData,Cnv>
CGAL::Arr_consolidated_curve_data_traits_2<Traits,Data>

See Also

ArrangementBasicTraits_2(page 2255)

ArrTraits::Intersect_2

Concept

Refines

Functor

Has Models

ArrangementXMonotoneTraits_2::Intersect_2

Output_iterator	*fo.operator()(*	*ArrTraits::X_monotone_curve_2 xc1,* *ArrTraits::X_monotone_curve_2 xc2,* *Output_iterator& oi)*

computes the intersections of *xc1* and *xc2* and inserts them in an ascending lexicographic *xy*-order into the output iterator *oi*. The value-type of *Output_iterator* is *CGAL::Object*, where each *Object* wraps either a *pair<ArrTraits::Point_2,ArrTraits::Multiplicity>* object, which represents an intersection point with its multiplicity (in case the multiplicity is undefined or unknown, it should be set to 0) or an *ArrTraits::X_monotone_curve_2* object, representing an overlapping subcurve of *xc1* and *xc2*. The operator returns a past-the-end iterator for the output sequence.

Concept

ArrTraits::Split_2

Refines

Functor

Has Models

ArrangementXMonotoneTraits_2::Split_2

void *fo.operator()(ArrTraits::X_monotone_curve_2 xc,*
ArrTraits::Point_2 p,
ArrTraits::X_monotone_curve_2& xc1,
ArrTraits::X_monotone_curve_2& xc2)

accepts an input curve *xc* and a split point *p* in its interior. It splits *xc* at the split point into two subcurves *xc1* and *xc2*, such that *p* is *xc1*'s *right* endpoint and *xc2*'s *left* endpoint.

ArrTraits::AreMergeable_2

Concept

Refines

Functor

Has Models

ArrangementXMonotoneTraits_2::Are_mergeable_2

bool	*fo(ArrTraits::X_monotone_curve_2 xc1, ArrTraits::X_monotone_curve_2 xc2)*
	accepts two x-monotone curves *xc1* and *xc2* and determines whether they can be merged to form a single x-monotone curve. *xc1* and *xc2* are mergeable if their underlying curves are identical, they share a common endpoint, and they do not bend to form a non-x-monotone curve.

Concept

ArrTraits::Merge_2

Refines

Functor

Has Models

ArrangementXMonotoneTraits_2::Merge_2

void *fo.merge(ArrTraits::X_monotone_curve_2 xc1,*
ArrTraits::X_monotone_curve_2 xc2,
ArrTraits::X_monotone_curve_2& xc)

accepts two *mergeable* x-monotone curves *xc1* and *xc2* and asigns *xc* with the merged curve.
Precondition: *are_mergeable_2*(*xc1*, *xc2*) is true.

ArrangementTraits_2

Definition

The concept ArrangementTraits_2 allows the construction of arrangement of *general* planar curves. Models of this concept are used by the free *insert()* functions of the arrangement package and by the *Arrangement_with_history_2* class.

A model of this concept must define the nested *Curve_2* type, which represents a general planar curve that is not necessarily x-monotone and is not necessarily connected. Such curves are eventually subdivided into x-monotone subcurves and isolated points (represented by the *Point_2* and *X_monotone_curve_2* types, defined in the basic traits concept).

A model of the concept ArrangementTraits_2 that handles arbitrary curves, which are always x-monotone, such as a traits class that handles linear curves may define the nested types *Curve_2* and *X_monotone_curve_2* to be of equivalent types. Moreover, defining them as of equivalent types is advantageous, as it enables a generic simple implementation of the nested Functor *Make_x_monotone_2*.

On the other hand, a model of the ArrangementTraits_2 concept that handles arbitrary curves, which may be not x-monotone must define the *Curve_2* and *X_monotone_curve_2* nested types to be of different types to allow proper dispatching of the free functions that accept such curves, such as *intsert()*.

Refines

ArrangementXMonotoneTraits_2

Types

ArrangementTraits_2:: Curve_2	models the concept *ArrTraits::Curve_2*.

Functor Types

ArrangementTraits_2:: Make_x_monotone_2

models the concept *ArrTraits::MakeXMonotone_2*.

Accessing Functor Objects

Make_x_monotone_2 *traits.make_x_monotone_2_object() const*

Has Models

CGAL::Arr_segment_traits_2<Kernel>
CGAL::Arr_non_caching_segment_traits_2<Kernel>
CGAL::Arr_linear_traits_2<Kernel>
CGAL::Arr_polyline_traits_2<SegmentTraits>

CGAL::Arr_circle_segment_traits_2<Kernel>
CGAL::Arr_line_arc_traits_2<CircularKernel>
CGAL::Arr_circular_arc_traits_2<CircularKernel>
CGAL::Arr_circular_line_arc_traits_2<CircularKernel>
CGAL::Arr_conic_traits_2<RatKernel,AlgKernel,NtTraits>
CGAL::Arr_rational_function_traits_2<AlgebraicKernel_d_1>
CGAL::Arr_Bezier_curve_traits_2<RatKernel,AlgKernel,NtTraits>
CGAL::Arr_algebraic_segment_traits_2<Coefficient>
CGAL::Arr_curve_data_traits_2<Tr,XData,Mrg,CData,Cnv>
CGAL::Arr_consolidated_curve_data_traits_2<Traits,Data>

See Also

ArrangementBasicTraits_2 (page 2255)
ArrangementXMonotoneTraits_2 (page 2273)
ArrangementLandmarkTraits_2 (page 2269)

ArrTraits::Curve_2

Definition

represents a general planar curve.

Refines

DefaultConstructible, CopyConstructible, Assignable

Has Models

ArrangementTraits_2::Curve_2

ArrTraits::MakeXMonotone_2

Concept

Refines

Functor

Has Models

ArrangementTraits_2::Make_x_monotone_2

template <typename OutputIterator>
OutputIterator *fo(ArrTraits::Curve_2 c, OutputIterator oi)*

subdivides the input curve c into x-monotone subcurves and isolated points, and inserts the results into a container through the given output iterator. The value type of *OutputIterator* is *CGAL::Object*, where each *Object* wraps either an *ArrTraits::X_monotone_curve_2* object or a *ArrTraits::Point_2* object. The operator returns a past-the-end iterator for the output sequence.

CGAL::Arr_segment_traits_2<Kernel>

Definition

The traits class *Arr_segment_traits_2<Kernel>* is a model of the *ArrangementTraits_2* concept, which allows the construction and maintenance of arrangements of line segments. It should be parameterized with a CGAL-kernel model that is templated in turn with a number type. To avoid numerical errors and robustness problems, the number type should support exact rational arithmetic — that is, the number type should support the arithmetic operations $+$, $-$, $\times$ and $\div$ carried out without loss of precision.

For example, instantiating the traits template with kernels, such as *Cartesian<Quotient<MP_Float> >*, or *Homogeneous<Gmpz>*, ensures the exact and robust operation of the application. In particular, the *Cartesian<Gmpq>* achieves the fastest running times in most cases. Using other inexact number types (for example, instantiating the template with *Simple_cartesian<double>*) is at the user's own risk: Selecting an inexact number type usually leads to faster running time at the expense of possible robustness problems.

For optimal performance, we recommend instantiating the traits class with the default *Exact_predicates_exact_constructions_kernel* provided by CGAL. Using this kernel guarantees exactness and robustness, while it incurs only a minor overhead (in comparison to working with a fast, inexact number type) for most inputs.

Arr_segment_traits_2<Kernel> defines *Kernel::Point_2* as its point type. However, it does *not* define *Kernel::Segment_2* as its curve type, as one may expect. The reason is that the kernel segment is represented by its two endpoints only, while the traits class needs to store extra data with its segments, in order to efficiently operate on them. Nevertheless, the nested *X_monotone_curve_2* and *Curve_2* types (in this case both types refer to the same class, as *every* line segment is (weakly) x-monotone) can however be converted to the type *Kernel::Segment_2*.

Arr_segment_traits_2<Kernel> achieves faster running times than the *Arr_non_caching_segment_traits_2<Kernel>* traits-class, when arrangements with relatively many intersection points are constructed. It also allows for working with less accurate, yet computationally efficient number types, such as *Quotient<MP_Float>*, which represents floating-point numbers with an unbounded mantissa, but with a bounded exponent. Using this traits class is therefore highly recommended for almost all applications that rely on arrangements of line segments. On the other hand, *Arr_segment_traits_2<Kernel>* uses more space and stores extra data with each segment, so constructing arrangements of huge sets of non-intersecting segments (or segments that intersect very sparsely) could be more efficient with the *Arr_non_caching_segment_traits_2* traits-class.

While *Arr_segment_traits_2<Kernel>* models the concept *ArrangementDirectionalXMonotoneTraits_2*, the implementation of the *Arr_mergeable_2* operation does not enforce the input curves to have the same direction as a precondition. Moreover, *Arr_segment_traits_2<Kernel>* supports the merging of curves of opposite directions.

#include <CGAL/Arr_segment_traits_2.h>

Is Model for the Concepts

ArrangementTraits_2
ArrangementLandmarkTraits_2
ArrangementDirectionalXMonotoneTraits_2

Class

CGAL::Arr_non_caching_segment_basic_traits_2<Kernel>

Definition

The traits class *Arr_non_caching_segment_basic_traits_2<Kernel>* is a model of the *ArrangementTraits_2* concept that allow the construction and maintenance of arrangements of sets of pairwise interior-disjoint line segments. It is templated with a CGAL-Kernel model, and it is derived from it. This traits class is a thin layer above the parameterized kernel. It inherits the *Point_2* from the kernel and its *X_monotone_curve_2* type is defined as *Kernel::Segment_2*. Most traits-class functor are inherited from the kernel functor, and the traits class only supplies the necessary functors that are not provided by the kernel. The kernel is parameterized with a number type, which should support the arithmetic operations $+$, $-$ and $\times$ in an exact manner in order to avoid robustness problems. Using *Cartesian<MP_Float>* or *Cartesian<Gmpz>* are possible instantiations for the kernel. Using other (inexact) number types (for example, instantiating the template with *Simple_cartesian<double>*) is also possible, at the user's own risk.

#include <*CGAL/Arr_non_caching_segment_basic_traits_2.h*>

Is Model for the Concepts

ArrangementLandmarkTraits_2

CGAL::Arr_non_caching_segment_traits_2<Kernel>

Class

Definition

The traits class *Arr_non_caching_segment_traits_2<Kernel>* is a model of the *ArrangementTraits_2* concept that allows the construction and maintenance of arrangements of line segments. It is parameterized with a CGAL-Kernel type, and it is derived from it. This traits class is a thin layer above the parameterized kernel. It inherits the *Point_2* from the kernel and its *X_monotone_curve_2* and *Curve_2* types are both defined as *Kernel::Segment_2*. Most traits-class functor are inherited from the kernel functor, and the traits class only supplies the necessary functors that are not provided by the kernel. The kernel is parameterized with a number type, which should support exact rational arithmetic in order to avoid robustness problems, although other number types could be used at the user's own risk.

The traits-class implementation is very simple, yet may lead to a cascaded representation of intersection points with exponentially long bit-lengths, especially if the kernel is parameterized with a number type that does not perform normalization (e.g. *Quotient<MP_Float>*). The *Arr_segment_traits_2* traits class avoids this cascading problem, and should be the default choice for implementing arrangements of line segments. It is recommended to use *Arr_non_caching_segment_traits_2<Kernel>* only for very sparse arrangements of huge sets of input segments.

While *Arr_non_caching_segment_traits_2<Kernel>* models the concept *ArrangementDirectionalXMonotoneTraits_2*, the implementation of the *Arr_mergeable_2* operation does not enforce the input curves to have the same direction as a precondition. Moreover, *Arr_non_caching_segment_traits_2<Kernel>* supports the merging of curves of opposite directions.

#include <CGAL/Arr_non_caching_segment_traits_2.h>

Is Model for the Concepts

ArrangementTraits_2
ArrangementLandmarkTraits_2
ArrangementDirectionalXMonotoneTraits_2

Inherits From

Arr_non_caching_segment_basic_traits_2<Kernel>

See Also

Arr_segment_traits_2<Kernel>

Class

CGAL::Arr_linear_traits_2<Kernel>

Definition

The traits class *Arr_linear_traits_2<Kernel>* is a model of the *ArrangementTraits_2* concept that allows for the construction and maintenance of arrangements of linear objects which may be bounded (line segments) or unbounded (rays and lines). The traits class is parameterized with a CGAL-kernel model; see the reference page of *Arr_segment_traits_2<Kernel>* (page 2283) for further explanations and recommendations on choosing a kernel.

Arr_linear_traits_2<Kernel> defines *Kernel::Point_2* as its point type. The nested *X_monotone_curve_2* and *Curve_2* types defined by the traits class (as is the case with the various segment-traits classes, both types refer to the same class, as *every* linear object is (weakly) x-monotone), are constructible from a point, a line segment, a ray and from a line (objects of types *Kernel::Point_2*, *Kernel::Segment_2*, *Kernel::Ray_2* and *Kernel::Line_2*, respectively). On the other hand, when we are given a curve we can find out its actual type and convert it to the respective kernel object (say, to a *Kernel::Ray_2*).

#include <CGAL/Arr_linear_traits_2.h>

Is Model for the Concepts

ArrangementTraits_2
ArrangementLandmarkTraits_2

Class Arr_linear_traits_2<Kernel>::Curve_2

The *Curve_2* (and the *X_monotone_curve_2*) class nested within the linear-traits can represent all types of linear objects. The class supports the copy and default constructor and the assignment operator. In addition, the *operator<<* and *operator>>* for linear objects are defined for standard output and input streams.

Types

typedef typename Kernel::Point_2
Point_2;

typedef typename Kernel::Segment_2
Segment_2;

typedef typename Kernel::Ray_2
Ray_2;

typedef typename Kernel::Line_2
Line_2;

Creation

Arr_linear_traits_2<Kernel>::Curve_2 c(Point_2 p);

constructs an degenerate curve equivalent to the point *p*.

Arr_linear_traits_2<Kernel>::Curve_2 c(Segment_2 seg);

constructs an line segment.

Arr_linear_traits_2<Kernel>::Curve_2 c(Ray_2 ray);

constructs a ray.

Arr_linear_traits_2<Kernel>::Curve_2 c(Line_2 ln);

constructs a line.

Access Functions

bool	*c.is_point() const*	returns whether *c* is a point (a degenerate curve).
Point_2	*c.point() const*	returns a point object equivalent to *c*. *Precondition*: *c* is a point.
bool	*c.is_segment() const*	returns whether *c* is a segment (has two bounded endpoints).
Segment_2	*c.segment() const*	returns a segment object equivalent to *c*. *Precondition*: *c* is a segment.
bool	*c.is_ray() const*	returns whether *c* is a ray (has a bounded source point and its other end is unbounded).
Ray_2	*c.ray() const*	returns a ray object equivalent to *c*. *Precondition*: *c* is a ray.
bool	*c.is_line() const*	returns whether *c* is a line (has only unbounded ends).
Line_2	*c.line() const*	returns a line object equivalent to *c*. *Precondition*: *c* is a line.
Line_2	*c.supporting_line() const*	returns the supporting line of *c*. *Precondition*: *c* is not a point (a degenerate curve).
Point_2	*c.source() const*	returns *c*'s source point. *Precondition*: *c* is a point, a segment or a ray.
Point_2	*c.target() const*	returns *c*'s target point. *Precondition*: *c* is a point or a segment.

Class

CGAL::Arr_polyline_traits_2<SegmentTraits>

Definition

The traits class *Arr_polyline_traits_2<SegmentTraits>* is a model of the *ArrangementTraits_2* concept. It handles piecewise linear curves, commonly referred to as polylines. Each polyline is a chain of segments, where each two neighboring segments in the chain share a common endpoint. The traits class exploits the functionality of the *SegmentTraits* template-parameter to handle the segments that comprise the polyline curves.

The class instantiated for the template parameter *SegmentTraits* must be a model of the *ArrangementTraits_2* concept that handles line segments (e.g., *Arr_segment_traits_2<Kernel>* or *Arr_non_caching_segment_traits_2<Kernel>*, where the first alternative is recommended).

The number type used by the injected segment traits should support exact rational arithmetic (that is, the number type should support the arithmetic operations $+$, $-$, $\times$ and $\div$ that should be carried out without loss of precision), in order to avoid robustness problems, although other inexact number types could be used at the user's own risk.

#include <CGAL/Arr_polyline_traits_2.h>

Is Model for the Concepts

ArrangementTraits_2
ArrangementLandmarkTraits_2

Class Arr_polyline_traits_2<SegmentTraits>::Curve_2

The *Curve_2* class nested within the polyline traits is used to represent general continuous piecewise-linear curves (a polyline can be self-intersecting) and support their construction from any range of points.

The copy and default constructor as well as the assignment operator are provided for polyline curves. In addition, an *operator<<* for the curves is defined for standard output streams, and an *operator>>* for the curves is defined for standard input streams.

Types

Arr_polyline_traits_2<SegmentTraits>::Curve_2:: const_iterator

A bidirectional iterator that allows traversing the points that comprise a polyline curve.

Arr_polyline_traits_2<SegmentTraits>::Curve_2:: const_reverse_iterator

A bidirectional iterator that allows traversing the points that comprise a polyline curve.

Creation

Arr_polyline_traits_2<SegmentTraits>::Curve_2 pi;

default constructor that constructs an empty polyline.

template <class InputIterator>
Arr_polyline_traits_2<SegmentTraits>::Curve_2 pi(Iterator first, Iterator last);

constructs a polyline defined by the given range of points *[first, last)* (the value-type of *InputIterator* must be *SegmentTraits::Point_2*. If the range contains $(n+1)$ points labeled $(p_0, p_1, \ldots, p_n)$, the generated polyline consists of n segments, where the kth segment is defined by the endpoints $[p_{k-1}, p_k]$. The first point in the range is considered as the source point of the polyline while the last point is considered as its target.
Precondition: There are at least two points in the range.

Access Functions

size_t	*pi.points() const*	returns the number of points that comprise the polyline. Note that if there are n points in the polyline, it is comprised of $(n-1)$ segments.
const_iterator	*pi.begin() const*	returns an iterator pointing at the source point of the polyline.
const_iterator	*pi.end() const*	returns an iterator pointing after the end of the polyline.
const_iterator	*pi.rbegin() const*	returns an iterator pointing at the target point of the polyline.
const_iterator	*pi.rend() const*	returns an iterator pointing before the beginning of the polyline.
size_t	*pi.size() const*	returns the number of line segments comprising the polyline (equivalent to *pi.points() - 1*).
typename SegmentTraits::X_monotone_curve_2	*pi*[*size_t k*]	returns the kth segment of the polyline. *Precondition*: k is not greater or equal to *pi.size() - 1*.
Bbox_2	*pi.bbox() const*	return a bounding box of the polyline *pi*.

Operations

void *pi.push_back(Point_2 p)*

adds a new point to the polyline, which becomes the new target point of *pi*.

void	*pi.clear()*	resets the polyline.

Class Arr_polyline_traits_2<SegmentTraits>::X_monotone_curve_2

The *X_monotone_curve_2* class nested within the polyline traits is used to represent x-monotone piecewise linear curves. It inherits from the *Curve_2* type. It has a default constructor and a constructor from a range of points, just like the *Curve_2* class. However, there is precondition that the point range define an x-monotone polyline.

The points that define the x-monotone polyline are always stored in an ascending lexicographical xy-order, so their order may be reversed with respect to the input sequence. Also note that the x-monotonicity ensures that an x-monotone polyline is never self-intersecting (thus, a self-intersecting polyline will be subdivided to several interior-disjoint x-monotone subcurves).

See Also

Arr_segment_traits_2<Kernel>
Arr_non_caching_segment_traits_2<Kernel>

CGAL::Arr_circle_segment_traits_2<Kernel>

Definition

The class *Arr_circle_segment_traits_2<Kernel>* is a model of the *ArrangementTraits_2* concept and can be used to construct and maintain arrangements of circular arcs and line segments.

The traits class must be instantiated with a geometric kernel, such that the supporting circles of the circular arcs are of type *Kernel::Circle_2* and the supporting lines of the line segments are of type *Kernel::Line_2*. Thus, the coordinates of the center of supporting circles, and its squared radius are of type *Kernel::FT*, which should be an exact rational number-type; similarly, the coefficients of each supporting line $ax+by+c=0$ are also of type *Kernel::FT*. Note however that the intersection point between two such arcs do not have rational coordinates in general. For this reason, we do not require the endpoints of the input arcs and segments to have rational coordinates.

The nested *Point_2* type defined by the traits class is therefore *different* than the *Kernel::Point_2* type. Its coordinates are of type *CoordNT*, which an instantiation of *CGAL::Sqrt_extension<NT,ROOT>* where *NT* = *ROOT* = *Kernel::FT*. Moreover, the third and fourth (hidden) template parameters of *CGAL::Sqrt_extension< NT,ROOT>* are set to *CGAL::Tag_true*, which enables efficient comparison among different extensions.

For more details see the documentation of *CGAL::Sqrt_extension<NT,ROOT>* . . . . page 220

While *Arr_circle_segment_traits_2<Kernel>* models the concept *ArrangementDirectionalXMonotoneTraits_2*, the implementation of the *Arr_mergeable_2* operation does not enforce the input curves to have the same direction as a precondition. Moreover, *Arr_circle_segment_traits_2<Kernel>* supports the merging of curves of opposite directions.

#include <CGAL/Arr_circle_segment_traits_2.h>

Is Model for the Concepts

ArrangementTraits_2
ArrangementDirectionalXMonotoneTraits_2

Class Arr_circle_segment_traits_2<Kernel>::Point_2

The *Point_2* number-type nested within the traits class represents a Cartesian point whose coordinates are algebraic numbers of type *CoordNT*.

Types

Arr_circle_segment_traits_2<Kernel>::Point_2:: Rational

the *Kernel::FT* type.

Arr_circle_segment_traits_2<Kernel>::Point_2:: CoordNT

the algebraic number-type.

Creation

Arr_circle_segment_traits_2<Kernel>::Point_2 p;

default constructor.

Arr_circle_segment_traits_2<Kernel>::Point_2 p(Rational x, Rational y);

creates the point (x,y).

Arr_circle_segment_traits_2<Kernel>::Point_2 p(CoordNT x, CoordNT y);

creates the point (x,y).

Access Functions

CoordNT	*p.x() const*	returns the *x*-coordinate.
CoordNT	*p.y() const*	returns the *y*-coordinate.

Class Arr_circle_segment_traits_2<Kernel>::Curve_2

The *Curve_2* class nested within the traits class can represent arbitrary circular arcs, full circles and line segments and support their construction in various ways. The copy and default constructor as well as the assignment operator are provided. In addition, an *operator<<* for the curves is defined for standard output streams.

Creation

Arr_circle_segment_traits_2<Kernel>::Curve_2 cv(typename Kernel::Segment_2 seg);

constructs an curve corresponding to the line segment *seg*.

Arr_circle_segment_traits_2<Kernel>::Curve_2 cv(typename Kernel::Point_2 source, typename Kernel::Point_2 target)

constructs an curve corresponding to the line segment directed from *source* to *target*, both having rational coordinates.

Arr_circle_segment_traits_2<Kernel>::Curve_2 cv(typename Kernel::Line_2 line, Point_2 source, Point_2 target)

constructs an curve corresponding to the line segment supported by the given line, directed from *source* to *target*. Note that the two endpoints may have one-root coordinates.
Precondition: Both endpoints must lie on the given supporting line.

Arr_circle_segment_traits_2<Kernel>::Curve_2 cv(typename Kernel::Circle_2 circ);

constructs an curve corresponding to the given circle. *circ* has a center point with rational coordinates and its *squared* radius is rational.

Arr_circle_segment_traits_2<Kernel>::Curve_2 cv(typename Kernel::Point_2 c,
typename Kernel::FT r,
Orientation orient = COUNTERCLOCKWISE)

constructs an curve corresponding to a circle centered at the rational point *c* whose radius *r* is rational.

Arr_circle_segment_traits_2<Kernel>::Curve_2 cv(typename Kernel::Circle_2 circ,
Point_2 source,
Point_2 target)

constructs a circular arc supported by *circ*, which has a center point with rational coordinates and whose *squared* radius is rational, with the given endpoints. The orientation of the arc is the same as the orientation of *circ*.
Precondition: Both endpoints must lie on the given supporting circle.

Arr_circle_segment_traits_2<Kernel>::Curve_2 cv(typename Kernel::Point_2 c,
typename Kernel::FT r,
Orientation orient,
Point_2 source,
Point_2 target)

constructs a circular arc supported by a circle centered at the rational point *c* whose radius *r* is rational, directed from *source* to *target* with the given orientation.
Precondition: Both endpoints must lie on the supporting circle.

Arr_circle_segment_traits_2<Kernel>::Curve_2 cv(typename Kernel::Point_2 source,
typename Kernel::Point_2 mid,
typename Kernel::Point_2 target)

constructs an circular arc whose endpoints are *source* and *target* that passes through *mid*. All three points have rational coordinates.
Precondition: The three points must not be collinear.

Access Functions

bool	*cv.is_full() const*	indicates whether the curve represents a full circle.
Point_2	*cv.source() const*	returns the source point. *Precondition*: *cv* is not a full circle.

Point_2	*cv.target() const*	returns the target point. *Precondition*: *cv* is not a full circle.
Orientation	*cv.orientation() const*	returns the orientation of the curve (*COLLINEAR* in case of line segments).
bool	*cv.is_linear() const*	determines whether *cv* is a line segment.
bool	*cv.is_circular() const*	determines whether *cv* is a circular arc.

typename Kernel::Line_2

cv.supporting_line() const

returns the supporting line of *cv*.
Precondition: *cv* is a line segment.

typename Kernel::Circle_2

cv.supporting_circle() const

returns the supporting circle of *cv*.
Precondition: *cv* is a circular arc.

Class Arr_circle_segment_traits_2<Kernel>::X_monotone_curve_2

The *X_monotone_curve_2* class nested within the traits class can represent *x*-monotone and line segments (which are always weakly *x*-monotone). The copy and default constructor as well as the assignment operator are provided. In addition, an *operator<<* for the curves is defined for standard output streams.

Creation

Arr_circle_segment_traits_2<Kernel>::X_monotone_curve_2 xcv(typename Kernel::Point_2 source, typename Kernel::Point_2 target)

constructs an curve corresponding to the line segment directed from *source* to *target*, both having rational coordinates.

Arr_circle_segment_traits_2<Kernel>::X_monotone_curve_2 xcv(typename Kernel::Line_2 line, Point_2 source, Point_2 target)

constructs an curve corresponding to the line segment supported by the given line, directed from *source* to *target*. Note that the two endpoints may have one-root coordinates.
Precondition: Both endpoints must lie on the given supporting line.

Arr_circle_segment_traits_2<Kernel>::X_monotone_curve_2 xcv(typename Kernel::Circle_2 circ,

Point_2 source,
Point_2 target,
Orientation orient)

constructs a circular arc supported by *circ*, which has a center point with rational coordinates and whose *squared* radius is rational, with the given endpoints. The orientation of the arc is determined by *orient*.
Precondition: Both endpoints must lie on the given supporting circle.
Precondition: The circular arc is x-monotone.

Access Functions

Point_2	*xcv.source() const*	returns the source point of *xcv*.
Point_2	*xcv.target() const*	returns the target point of *xcv*.
bool	*xcv.is_directed_right() const*	returns true if *xcv* is directed right, false otherwise.
Point_2	*xcv.left() const*	returns the left (lexicographically smaller) endpoint of *xcv*.
Point_2	*xcv.right() const*	returns the right (lexicographically larger) endpoint of *xcv*.
Orientation	*xcv.orientation() const*	returns the orientation of the curve (*COLLINEAR* in case of line segments).
bool	*xcv.is_linear() const*	determines whether *xcv* is a line segment.
bool	*xcv.is_circular() const*	determines whether *xcv* is a circular arc.
typename Kernel::Line_2	*xcv.supporting_line() const*	returns the supporting line of *xcv*. *Precondition*: *xcv* is a line segment.
typename Kernel::Circle_2	*xcv.supporting_circle() const*	returns the supporting circle of *xcv*. *Precondition*: *xcv* is a circular arc.
Bbox_2	*xcv.bbox() const*	returns a bounding box of the arc *xcv*.

Class

CGAL::Arr_circular_arc_traits_2<CircularKernel>

#include <CGAL/Arr_circular_arc_traits_2.h>

Definition

This class is a traits class for CGAL arrangements, built on top of a model of concept *CircularKernel*. It provides curves of type *CGAL::Circular_arc_2<CircularKernel>*.

Is Model for the Concepts

ArrangementTraits_2

CGAL::Arr_line_arc_traits_2<CircularKernel>

#include <CGAL/Arr_line_arc_traits_2.h>

Class

Definition

This class is a traits class for CGAL arrangements, built on top of a model of concept *CircularKernel*. It provides curves of type *CGAL::Line_arc_2<CircularKernel>*.

Is Model for the Concepts

ArrangementTraits_2

Class

CGAL::Arr_circular_line_arc_traits_2<CircularKernel>

#include <CGAL/Arr_circular_line_arc_traits_2.h>

Definition

This class is a traits class for CGAL arrangements, built on top of a model of concept *CircularKernel*. It provides curves that can be of both types *CGAL::Line_arc_2<CircularKernel>* or *CGAL::Circular_arc_2<CircularKernel>*.

It uses the boost::variant.

Is Model for the Concepts

ArrangementTraits_2

CGAL::Arr_conic_traits_2<RatKernel,AlgKernel,NtTraits>

Class

Definition

The class *Arr_conic_traits_2<RatKernel,AlgKernel,NtTraits>* is a model of the *ArrangementTraits_2* concept and can be used to construct and maintain arrangements of bounded segments of algebraic curves of degree 2 at most, also known as *conic curves*.

A general conic curve C is the locus of all points (x,y) satisfying the equation: $rx^2+sy^2+txy+ux+vy+w=0$, where:

- If $4rs-t^2>0$, C is an ellipse. A special case occurs when $r=s$ and $t=0$, when C becomes a circle.
- If $4rs-t^2<0$, C is a hyperbola.
- If $4rs-t^2=0$, C is a parabola. A degenerate case occurs when $r=s=t=0$, when C is a line.

A *bounded conic arc* is defined as either of the following:

- A full ellipse (or a circle) C.
- The tuple $\langle C, p_s, p_t, o\rangle$, where C is the supporting conic curve, with the arc endpoints being p_s and p_t (the source and target points, respectively). The orientation o indicates whether we proceed from p_s to p_t in a clockwise or in a counterclockwise direction. Note that C may also correspond to a line or to pair of lines — in this case o may specify a *COLLINEAR* orientation.

A very useful subset of the set of conic arcs are line segments and circular arcs, as arrangements of circular arcs and line segments have some interesting applications (e.g. offsetting polygons, motion planning for a disc robot, etc.). Circular arcs and line segment are simpler objects and can be dealt with more efficiently than arbitrary arcs. For these reasons, it is possible to construct conic arcs from segments and from circles. Using these constructors is highly recommended: It is more straightforward and also speeds up the arrangement construction. However, in case the set of input curves contain only circular arcs and line segments, it is recommended to use the *Arr_circle_segment_2* class to achieve faster running times.

In our representation, all conic coefficients (namely r,s,t,u,v,w) must be rational numbers. This guarantees that the coordinates of all arrangement vertices (in particular, those representing intersection points) are algebraic numbers of degree 4 (a real number α is an algebraic number of degree d if there exist a polynomial p with *integer* coefficient of degree d such that $p(\alpha)=0$). We therefore require separate representations of the curve coefficients and the point coordinates. The *NtTraits* should be instantiated with a class that defines nested *Integer*, *Rational* and *Algebraic* number types and supports various operations on them, yielding certified computation results (for example, it can convert rational numbers to algebraic numbers and can compute roots of polynomials with integer coefficients). The other template parameters, *RatKernel* and *AlgKernel* should be geometric kernels templated with the *NtTraits::Rational* and *NtTraits::Algebraic* number types, respectively. It is recommended to instantiate the *CORE_algebraic_number_traits* class as the *NtTraits* parameter, with *Cartesian<NtTraits::Rational>* and *Cartesian<NtTraits::Algebraic>* instantiating the two kernel types, respectively. The number types in this case are provided by the CORE library, with its ability to exactly represent simple algebraic numbers.

The traits class inherits its point type from *AlgKernel::Point_2*, and defines a curve and x-monotone curve types, as detailed below.

While the *Arr_conic_traits_2<RatKernel,AlgKernel,NtTraits>* models the concept *ArrangementDirectionalXMonotoneTraits_2*, the implementation of the *Arr_mergeable_2* operation does not enforce the input curves to have the same direction as a precondition. Moreover, *Arr_conic_traits_2<RatKernel,AlgKernel,NtTraits>* supports the merging of curves of opposite directions.

#include <CGAL/Arr_conic_traits_2.h>

Is Model for the Concepts

ArrangementTraits_2
ArrangementLandmarkTraits_2
ArrangementDirectionalXMonotoneTraits_2

Types

Arr_conic_traits_2<RatKernel,AlgKernel,NtTraits>:: Rational

the *NtTraits::Rational* type (and also the *RatKernel::FT* type).

Arr_conic_traits_2<RatKernel,AlgKernel,NtTraits>:: Algebraic

the *NtTraits::Algebraic* type (and also the *AlgKernel::FT* type).

Class Arr_conic_traits_2<RatKernel,AlgKernel,NtTraits>::Curve_2

The *Curve_2* class nested within the conic-arc traits can represent arbitrary conic arcs and support their construction in various ways. The copy and default constructor as well as the assignment operator are provided for conic arcs. In addition, an *operator<<* for the curves is defined for standard output streams.

Creation

Arr_conic_traits_2<RatKernel,AlgKernel,NtTraits>::Curve_2 a(typename RatKernel::Segment_2 seg);

constructs an arc corresponding to the line segment *seg*.

Arr_conic_traits_2<RatKernel,AlgKernel,NtTraits>::Curve_2 a(typename RatKernel::Circle_2 circ);

constructs an arc corresponding to the full circle *circ* (note that this circle has a center point with rational coordinates and rational squared radius).

Arr_conic_traits_2<RatKernel,AlgKernel,NtTraits>::Curve_2 a(typename RatKernel::Circle_2 circ,
Orientation o,
Point_2 ps,

Point_2 pt)

constructs a circular arc supported by the circle *circ*, going in the given orientation *o* from the source point *ps* to its target point *pt*.
Precondition: *ps* and *pt* both lie on the circle *circ*.
Precondition: *o* is not *COLLINEAR*.

Arr_conic_traits_2<RatKernel,AlgKernel,NtTraits>::Curve_2 a(typename RatKernel::Point_2 p1, typename RatKernel::Point_2 p2, typename RatKernel::Point_2 p3)

constructs a circular arc going from *p1* (its source point) through *p2* to *p3* (its target point). Note that all three points have rational coordinates. The orientation of the arc is determined automatically.
Precondition: The three points are not collinear.

Arr_conic_traits_2<RatKernel,AlgKernel,NtTraits>::Curve_2 a(Rational r, Rational s, Rational t, Rational u, Rational v, Rational w)

constructs a conic arc that corresponds to the full conic curve $rx^2 + sy^2 + txy + ux + vy + w = 0$.
Precondition: As a conic arc must be bounded, the given curve must be an ellipse, that is $4rs - t^2 > 0$.

Arr_conic_traits_2<RatKernel,AlgKernel,NtTraits>::Curve_2 a(Rational r, Rational s, Rational t, Rational u, Rational v, Rational w, Orientation o, Point_2 ps, Point_2 pt)

constructs a conic arc supported by the conic curve $rx^2 + sy^2 + txy + ux + vy + w = 0$, going in the given orientation *o* from the source point *ps* to its target point *pt*.
Precondition: *ps* and *pt* both satisfy the equation of the supporting conic curve and define a bounded segment of this curve (e.g. in case of a hyperbolic arc, both point should be located on the same branch of the hyperbola).
Precondition: *o* is not *COLLINEAR* if the supporting conic is curves, and must be *COLLINEAR* if it is not curved (a line or a line-pair).

Arr_conic_traits_2<RatKernel,AlgKernel,NtTraits>::Curve_2 a(typename RatKernel::Point_2 p1,

typename RatKernel::Point_2 p2,
typename RatKernel::Point_2 p3,
typename RatKernel::Point_2 p4,
typename RatKernel::Point_2 p5)

constructs a conic arc going from *p1* (its source point) through *p2*, *p3* and *p4* (in this order) to *p5* (its target point). Note that all five points have rational coordinates. The orientation of the arc is determined automatically.
Precondition: No three points of the five are not collinear.
Precondition: The five points define a valid arc, in their given order.

Arr_conic_traits_2<RatKernel,AlgKernel,NtTraits>::Curve_2 a(Rational r,
Rational s,
Rational t,
Rational u,
Rational v,
Rational w,
Orientation o,
Point_2 app_ps,
Rational r1,
Rational s1,
Rational t1,
Rational u1,
Rational v1,
Rational w1,
Point_2 app_pt,
Rational r2,
Rational s2,
Rational t2,
Rational u2,
Rational v2,
Rational w2)

constructs a conic arc supported by the conic curve $rx^2 + sy^2 + txy + ux + vy + w = 0$, going in the given orientation o from its source point to its target point. In this case only some approximations of the endpoints (*app_ps* and *app_pt*, respectively) is available, and their exact locations are given implicitly, specified by the intersections of the supporting conic curve with $r_1x^2 + s_1y^2 + t_1xy + u_1x + v_1y + w_1 = 0$ and $r_2x^2 + s_2y^2 + t_2xy + u_2x + v_2y + w_2 = 0$, respectively.
Precondition: The two auxiliary curves specifying the endpoints really intersect with the supporting conic curve, such that the arc endpoints define a bounded segment of the supporting curve (e.g. in case of a hyperbolic arc, both point should be located on the same branch of the hyperbola).
Precondition: o is not *COLLINEAR* if the supporting conic is curves, and must be *COLLINEAR* if it is not curved (a line or a line-pair).

Access Functions

bool	*a.is_valid() const*	indicates whether a is a valid conic arc. As the precondition to some of the constructor listed above are quite complicated, their violation does not cause the program to abort. Instead, the constructed arc is invalid (a defaultly constructed arc is also invalid). It is however recommended to check that a constructed arc is valid before inserting it to an arrangement, as this operation *will* cause the program to abort.
bool	*a.is_x_monotone() const*	determines whether the arc is x-monotone, namely each vertical line intersects it at most once. A vertical line segment is also considered (weakly) x-monotone.
bool	*a.is_y_monotone() const*	determines whether the arc is y-monotone, namely each horizontal line intersects it at most once. A horizontal line segment is also considered (weakly) x-monotone.
bool	*a.is_full_conic() const*	indicates whether the arc represents a full conic curve (en ellipse or a circle).

The six following methods return the coefficients of the supported conic, after their conversion to integer number (represented by the *Integer* type of the *NtTraits* class):

typename NtTraits::Integer	*a.r() const*	returns the coefficient of x^2.
typename NtTraits::Integer	*a.s() const*	returns the coefficient of t^2.
typename NtTraits::Integer	*a.t() const*	returns the coefficient of xy.
typename NtTraits::Integer	*a.u() const*	returns the coefficient of x.
typename NtTraits::Integer	*a.v() const*	returns the coefficient of y.
typename NtTraits::Integer	*a.w() const*	returns the free coefficient.

Point_2	*a.source() const*	returns the source point of the arc. *Precondition*: a is not a full conic curve.
Point_2	*a.target() const*	returns the target point of the arc. *Precondition*: a is not a full conic curve.
Orientation	*a.orientation() const*	returns the orientation of the arc.
Bbox_2	*a.bbox() const*	return a bounding box of the arc a.

Operations

void	*a.set_source(Point_2 ps)*	sets a new source point for the conic arc. *Precondition*: ps lies on the supporting conic of a.
void	*a.set_target(Point_2 pt)*	sets a new target point for the conic arc. *Precondition*: pt lies on the supporting conic of a.

Class Arr_conic_traits_2<RatKernel,AlgKernel,NtTraits>::X_monotone_curve_2

The *X_monotone_curve_2* class nested within the conic-arc traits is used to represent x-monotone conic arcs. It inherits from the *Curve_2* type, therefore supports the access methods and the operations listed above.

For efficiency reasons, we recommend users not to construct x-monotone conic arc directly, but rather use the *Make_x_monotone_2* functor supplied by the conic-arc traits class to convert conic curves to x-monotone curves.

Creation

Arr_conic_traits_2<RatKernel,AlgKernel,NtTraits>:: X_monotone_curve_2 xa(Curve_2 arc);

converts the given arc to an x-monotone arc.
Precondition: *arc* is x-monotone.

Access Functions

Point_2	*xa.left() const*	returns the left (lexicographically smaller) endpoint of *xa*.
Point_2	*xa.right() const*	returns the right (lexicographically larger) endpoint of *xa*.

CGAL::Arr_rational_function_traits_2<AlgebraicKernel_d_1>

Class

Definition

The traits class *Arr_rational_function_traits_2<AlgebraicKernel_d_1>* is a model of the *ArrangementTraits_2* concept. It handles bounded and unbounded arcs of rational functions, referred to as rational arcs (in particular, such an arc may correspond to the entire graph of a rational function), and enables the construction and maintenance of arrangements of such arcs.

A rational function $y = \frac{P(x)}{Q(x)}$ is defined by two polynomials P and Q of arbitrary degrees. If $Q(x) = 1$ then the function is a simple polynomial function. Usually the domain is $\mathbb{R}$ but the function may also be restricted to a bounded interval $[x_{\min}, x_{\max}]$ or defined over a ray $(-\infty, x_{\max}]$ or over $[x_{\min}, \infty)$. Rational functions are represented by the nested type *Curve_2*. Note that a rational function may be not continuous since roots of Q induce vertical asymptotes, which would contradict the notion of an x-monotone curve as it is introduced by the *ArrangementTraits_2* concept. Thus, continuous portions of rational functions are represented by the nested type *X_monotone_curve_2*, which is different from *Curve_2*. Constructors for both classes are provided by the traits. A *Curve_2* may be split up into several *X_monotone_curve_2* using *Make_x_monotone_2*.

The template parameter of the traits must be a model of the concept *AlgebraicKernel_d_1*. A rational function is then represented by two polynomials P and Q of type *AlgebraicKernel_d_1::Polynomial_1*. A point is represented by a rational function and its x-coordinate, which is of type *AlgebraicKernel_d_1::Algebraic_real_1*. Note that an explicit representation of the y-coordinate is only computed upon request, which can be a rather costly operation.

The constructed rational functions are cached by the traits class. The cache is local to each traits class object. It is therefore necessary to construct the curves using the constructor objects provided by member functions of the traits class. Moreover, a curve must only be used with its own traits. The cache is automatically cleaned up from time to time. The amortized clean up costs are constant. However, there is also a separate member function that cleans up the cache on demand.

While *Arr_rational_function_traits_2<AlgebraicKernel_d_1>* models the concept *ArrangementDirectionalXMonotoneTraits_2*, the implementation of the *Arr_mergeable_2* operation does not enforce the input curves to have the same direction as a precondition. Moreover, *Arr_rational_function_traits_2<AlgebraicKernel_d_1>* supports the merging of curves of opposite directions.

#include <CGAL/Arr_rational_function_traits_2.h>

Is Model for the Concepts

ArrangementTraits_2
ArrangementDirectionalXMonotoneTraits_2
ArrangementOpenBoundaryTraits_2

Types

typedef AlgebraicKernel_d_1	*Algebraic_kernel_d_1;*
typedef AlgebraicKernel_d_1::Coefficient	*Coefficient;*
typedef AlgebraicKernel_d_1::Polynomial_1	*Polynomial_1;*
typedef AlgebraicKernel_d_1::Algebraic_real_1	*Algebraic_real_1;*
typedef AlgebraicKernel_d_1::Bound	*Bound;*

Creation

Arr_rational_function_traits_2<AlgebraicKernel_d_1> traits(const Algebraic_kernel_d_1 kernel);*

constructs an empty traits that uses the kernel pointed by *kernel* for performing algebraic operations.

Operations

Construct_curve_2	*traits.construct_curve_2_object() const*	
		Returns an instance of *Construct_curve_2*.
Construct_x_monotone_curve_2	*traits.construct_x_monotone_curve_2_object() const*	
		Returns an instance of *Construct_x_monotone_curve_2*.
void	*traits.cleanup_cache() const*	
		Deletes all curves from the cache that exist only there.
*const Algebraic_kernel_d_1**	*traits.algebraic_kernel_d_1() const*	
		Returns a pointer to the used algerbaic kernel object.

Class Arr_rational_function_traits_2<AlgebraicKernel_d_1>::Curve_2

The *Curve_2* class nested within the traits is used to represent rational functions which may be restricted to a certain x-range.

Is Model for the Concepts

ArrTraits::Curve_2

Types

typedef AlgebraicKernel_d_1::Polynomial_1	*Polynomial_1;*
typedef AlgebraicKernel_d_1::Algebraic_real_1	*Algebraic_real_1;*

Operations

Polynomial_1	*curve.numerator() const*	returns the numerator of the supporting rational function.
Polynomial_1	*curve.denominator() const*	returns the denominator of the supporting rational function.

bool	*curve.is_continuous() const*	returns whether *curve* is continuous, namely whether it does not contains any vertical asymptotes in its interior.
Arr_parameter_space	*curve.left_parameter_space_in_x() const*	
		returns whether the x-coordinate of *curve*'s left end is finite or whether it is $\pm\infty$.
Arr_parameter_space	*curve.right_parameter_space_in_x() const*	
		returns whether the x-coordinate of *curve*'s right end is finite or whether it is $\pm\infty$.
Algebraic_real_1	*curve.left_x() const*	returns the x-coordinate of the left end. *Precondition*: left_boundary_in_x()==ARR_INTERIOR
Algebraic_real_1	*curve.right_x() const*	returns the x-coordinate of the right end. *Precondition*: right_boundary_in_x()==ARR_INTERIOR

Class Arr_rational_function_traits_2<AlgebraicKernel_d_1>::X_monotone_curve_2

The *X_monotone_curve_2* class nested within the traits is used to represent x-monotone parts of rational functions. In particular, such an x-monotone curve may not contain a vertical asymptote in its interior x-range.

Is Model for the Concepts

ArrTraits::XMonotoneCurve_2

Types

typedef AlgebraicKernel_d_1::Polynomial_1	*Polynomial_1;*
typedef AlgebraicKernel_d_1::Algebraic_real_1	*Algebraic_real_1;*
typedef Arr_rational_function_traits_2<AlgebraicKernel_d_1>::Point_2	*Point_2;*

Operations

Polynomial_1	*xcurve.numerator() const*	returns the numerator of the supporting rational function.
Polynomial_1	*xcurve.denominator() const*	returns the denominator of the supporting rational function.
Arr_parameter_space	*xcurve.source_parameter_space_in_x() const*	
		returns whether the x-coordinate of the source is finite or whether it is $\pm\infty$.
Arr_parameter_space	*xcurve.source_parameter_space_in_y() const*	
		returns whether the y-coordinate of the source is finite or whether it is $\pm\infty$.

Point_2	*xcurve.source() const*	returns the source point of the arc. *Precondition*: Both the x- and y-coordinates of the source point is finite.
Algebraic_real_1	*xcurve.source_x() const*	returns the x-coordinate of the source point. *Precondition*: The x-coordinate of the source point is finite.
Arr_parameter_space	*xcurve.target_parameter_space_in_x() const*	
		returns whether the x-coordinate of the target is finite or whether it is $\pm\infty$.
Arr_parameter_space	*xcurve.target_parameter_space_in_y() const*	
		returns whether the y-coordinate of the target is finite or whether it is $\pm\infty$.
Point_2	*xcurve.target() const*	returns the target point of the arc. *Precondition*: Both the x- and y-coordinates of the target point is finite.
Algebraic_real_1	*xcurve.target_x() const*	returns the x-coordinate of the target point. *Precondition*: The x-coordinate of the target point is finite.
Arr_parameter_space	*xcurve.left_parameter_space_in_x() const*	
		returns whether the x-coordinate of the left curve end is finite or whether it is $\pm\infty$.
Arr_parameter_space	*xcurve.left_parameter_space_in_y() const*	
		returns whether the y-coordinate of the left curve end is finite or whether it is $\pm\infty$.
Point_2	*xcurve.left() const*	returns the left point of the arc. *Precondition*: Both the x- and y-coordinates of the left point is finite.
Algebraic_real_1	*xcurve.left_x() const*	returns the x-coordinate of the left point. *Precondition*: The x-coordinate of the left point is finite.
Arr_parameter_space	*xcurve.right_parameter_space_in_x() const*	
		returns whether the x-coordinate of the right curve end is finite or whether it is $\pm\infty$.
Arr_parameter_space	*xcurve.right_parameter_space_in_y() const*	
		returns whether the y-coordinate of the right curve end is finite or whether it is $\pm\infty$.
Point_2	*xcurve.right() const*	returns the right point of the arc. *Precondition*: Both the x- and y-coordinates of The right point is finite.
Algebraic_real_1	*xcurve.right_x() const*	returns the x-coordinate of the right point. *Precondition*: The x-coordinate of the right point is finite.
bool	*xcurve.is_left_to_right() const*	returns whether the curve is oriented from left to right.

Class Arr_rational_function_traits_2<AlgebraicKernel_d_1>::Point_2

Is Model for the Concepts

ArrTraits::Point_2

Types

typedef AlgebraicKernel_d_1::Polynomial_1	*Polynomial_1;*
typedef AlgebraicKernel_d_1::Algebraic_real_1	*Algebraic_real_1;*
typedef AlgebraicKernel_d_1::Bound	*Bound;*

Operations

Polynomial_1	*point.numerator() const*	returns the numerator of the supporting rational function.
Polynomial_1	*point.denominator() const*	returns the denominator of the supporting rational function.
std::pair<double,double>	*point.to_double() const*	returns double-approximations of the x- and y-coordinates.
Algebraic_real_1	*point.x() const*	returns the x-coordinate of the point.

Algebraic_real_1 *point.y() const*

obtains the y-coordinates of the point. **Attention:** As described above, points are not stored by their y-coordinate in *Algebraic_real_1* representation. In fact, this representation must be computed on demand, and might become quite costly for points defined by high-degree polynomials. Therefore, it is recommended to avoid calls to this function as much as possible.

std::pair<Bound,Bound> *point.approximate_absolute_x(int a) const*

Computes a pair p approximating the x-coordinate with respect to the given absolute precision a.
Postcondition: $p.first \leq x \leq p.second$
Postcondition: $p.second - p.first \leq 2^{-a}$

std::pair<Bound,Bound> *point.approximate_absolute_y(int a) const*

Computes a pair p approximating the y-coordinate with respect to the given absolute precision a.
Postcondition: $p.first \leq y \leq p.second$
Postcondition: $p.second - p.first \leq 2^{-a}$

std::pair<Bound,Bound> *point.approximate_relative_x(int r) const*

Computes a pair p approximating the x-coordinate with respect to the given relative precision r.
Postcondition: $p.first \leq x \leq p.second$
Postcondition: $p.second - p.first \leq 2^{-r}|x|$

std::pair<Bound,Bound> *point.approximate_relative_y(int r) const*

Computes a pair p approximating the y-coordinate with respect to the given relative precision r.
Postcondition: $p.first \leq y \leq p.second$
Postcondition: $p.second - p.first \leq 2^{-r}|y|$

Class Arr_rational_function_traits_2<AlgebraicKernel_d_1>::Construct_curve_2

Functor to construct a *Curve_2*. To enable caching the class is not default constructible and must be obtained via the function *construct_curve_2_object()*, which is a member of the traits.

Is Model for the Concepts

Assignable
CopyConstructible
AdaptableBinaryFunction
AdaptableUnaryFunction

Types

typedef AlgebraicKernel_d_1::Polynomial_1 *Polynomial_1;*
typedef AlgebraicKernel_d_1::Algebraic_real_1 *Algebraic_real_1;*
typedef Arr_rational_function_traits_2<AlgebraicKernel_d_1>::Curve_2 *result_type;*

typedef Polynomial_1 *argument_type;*
typedef Polynomial_1 *first_argument_type;*
typedef Polynomial_1 *second_argument_type;*

Operations

Curve_2 *construct(Polynomial_1 P)*

Constructs a curve representing the polynomial function $y = P(x)$.

Curve_2 *construct(Polynomial_1 P, Algebraic_real_1 x, bool right)*

Constructs a curve representing the polynomial function $y = P(x)$. The function is defined over the interval $[x, +\infty)$ if *right* is true and $(-\infty, x]$ otherwise.

Curve_2 *construct(Polynomial_1 P, Algebraic_real_1 lower, Algebraic_real_1 upper)*

Constructs a curve representing the polynomial function $y = P(x)$. The function is defined over the interval $[lower, upper]$.

Curve_2 *construct(Polynomial_1 P, Polynomial_1 Q)*

Constructs a curve representing the rational function $y = P(x)/Q(x)$.

Curve_2 *construct(Polynomial_1 P, Polynomial_1 Q, Algebraic_real_1 x, bool right)*

Constructs a curve representing the rational function $y = P(x)/Q(x)$. The function is defined over the interval $I = [x, +\infty)$ if *right* is true and $I = (-\infty, x]$ otherwise.

Curve_2 *construct(Polynomial_1 P, Polynomial_1 Q, Algebraic_real_1 lower, Algebraic_real_1 upper)*

Constructs a curve representing the rational function $y = P(x)/Q(x)$. The function is defined over the interval $I = [lower, upper]$.

template <typename InputIterator>
Curve_2 *construct(InputIterator begin, InputIterator end)*

Constructs a curve representing the polynomial function $y = P(x)$, where the coefficients of P are given in the range *[begin,end)*.

template <typename InputIterator>
Curve_2 *construct(InputIterator begin, InputIterator end, Algebraic_real_1 x, bool right)*

Constructs a curve representing the polynomial function $y = P(x)$, where the coefficients of P are given in the range *[begin,end)*. The function is defined over the interval $[x, +\infty)$ if *right* is true and $(-\infty, x]$ otherwise.

template <typename InputIterator>
Curve_2 *construct.operator()(InputIterator begin,*
InputIterator end,
Algebraic_real_1 lower,
Algebraic_real_1 upper) const

Constructs a curve representing the polynomial function $y = P(x)$, where the coefficients of P are given in the range *[begin,end)*. The function is defined over the interval $[lower, upper]$.

template <typename InputIterator>
Curve_2 *construct.operator()(InputIterator begin_numer,*
InputIterator end_numer,
InputIterator begin_denom,
InputIterator end_denom) const

Constructs a curve representing the rational function $y = P(x)/Q(x)$, where the coefficients of P and Q are given in the ranges *[begin_numer,end_numer)* and *[begin_denom,end_denom)*, respectively.

template <typename InputIterator>
Curve_2 *construct.operator()(InputIterator begin_numer,*
InputIterator end_numer,
InputIterator begin_denom,
InputIterator end_denom,
Algebraic_real_1 x,
bool right) const

Constructs a curve representing the rational function $y = P(x)/Q(x)$, where the coefficients of P and Q are given in the ranges *[begin_numer,end_numer)* and *[begin_denom,end_denom)*, respectively. The function is defined over the interval $I = [x, +\infty)$ if *right* is true and $I = (-\infty, x]$ otherwise.

template <typename InputIterator>
Curve_2 *construct.operator()(InputIterator begin_numer, InputIterator end_numer, InputIterator begin_denom, InputIterator end_denom, Algebraic_real_1 lower, Algebraic_real_1 upper) const*

Constructs a curve representing the rational function $y = P(x)/Q(x)$, where the coefficients of P and Q are given in the ranges *[begin_numer,end_numer)* and *[begin_denom,end_denom)*, respectively. The function is defined over the interval $I = [lower, upper]$.

Class Arr_rational_function_traits_2<AlgebraicKernel_d_1>::Construct_x_monotone_curve_2

Functor to construct a *X_monotone_curve_2*. To enable caching the class is not default constructible and must be obtained via the function *construct_x_monotone_curve_2_object()*, which is a member of the traits.

Is Model for the Concepts

Assignable
CopyConstructible
AdaptableBinaryFunction
AdaptableUnaryFunction

Types

typedef AlgebraicKernel_d_1::Polynomial_1 *Polynomial_1;*
typedef AlgebraicKernel_d_1::Algebraic_real_1 *Algebraic_real_1;*
typedef Arr_rational_function_traits_2<AlgebraicKernel_d_1>::X_monotone_curve_2 *result_type;*
typedef Polynomial_1 *argument_type;*
typedef Polynomial_1 *first_argument_type;*
typedef Polynomial_1 *second_argument_type;*

Operations

X_monotone_curve_2 *construct(Polynomial_1 P)*

Constructs an x-monotone curve supported by the polynomial function $y = P(x)$.

X_monotone_curve_2 *construct(Polynomial_1 P, Algebraic_real_1 x, bool right)*

Constructs an x-monotone curve supported by the polynomial function $y = P(x)$. The function is defined over the interval $[x, +\infty)$ if *right* is true and $(-\infty, x]$ otherwise.

X_monotone_curve_2 *construct(Polynomial_1 P, Algebraic_real_1 lower, Algebraic_real_1 upper)*

Constructs an x-monotone curve supported by the polynomial function $y = P(x)$. The function is defined over the interval $[lower, upper]$.

X_monotone_curve_2 *construct(Polynomial_1 P, Polynomial_1 Q)*

Constructs an x-monotone curve supported by the rational function $y = P(x)/Q(x)$.
Precondition: Q has no real roots.

X_monotone_curve_2 *construct(Polynomial_1 P, Polynomial_1 Q, Algebraic_real_1 x, bool right)*

Constructs an x-monotone curve supported by the rational function $y = P(x)/Q(x)$. The function is defined over the interval $I = [x, +\infty)$ if *right* is true and $I = (-\infty, x]$ otherwise.
Precondition: Q has no real roots in the interior of I.

X_monotone_curve_2 *construct.operator()(Polynomial_1 P,*
Polynomial_1 Q,
Algebraic_real_1 lower,
Algebraic_real_1 upper)

Constructs an x-monotone curve supported by the rational function $y = P(x)/Q(x)$. The function is defined over the interval $I = [lower, upper]$.
Precondition: Q has no real roots in the interior of I.

template <typename InputIterator>
X_monotone_curve_2 *construct(InputIterator begin, InputIterator end)*

Constructs an x-monotone curve supported by the polynomial function $y = P(x)$, where the coefficients of P are given in the range *[begin,end)*.

template <typename InputIterator>
X_monotone_curve_2 *construct(InputIterator begin, InputIterator end, Algebraic_real_1 x, bool right)*

Constructs an x-monotone curve supported by the polynomial function $y = P(x)$, where the coefficients of P are given in the range *[begin,end)*. The function is defined over the interval $[x, +\infty)$ if *right* is true and $(-\infty, x]$ otherwise.

template <typename InputIterator>
X_monotone_curve_2 *construct.operator()(InputIterator begin,*
InputIterator end Algebraic_real_1 lower,
Algebraic_real_1 upper)

Constructs an x-monotone curve supported by the polynomial function $y = P(x)$, where the coefficients of P are given in the range *[begin,end)*. The function is defined over the interval $[lower, upper]$.

template <typename InputIterator>
X_monotone_curve_2 *construct.operator()(InputIterator begin_numer,*
InputIterator end_numer,
InputIterator begin_denom,
InputIterator end_denom)

Constructs an x-monotone curve supported by the rational function $y = P(x)/Q(x)$, where the coefficients of P and Q are given in the ranges *[begin_numer,end_numer)* and *[begin_denom,end_denom)*, respectively.
Precondition: Q has no real roots.

template <typename InputIterator>
X_monotone_curve_2 *construct.operator()(InputIterator begin_numer,*

InputIterator end_numer,
InputIterator begin_denom,
InputIterator end_denom,
Algebraic_real_1 x,
bool right)

Constructs an x-monotone curve supported by the rational function $y = P(x)/Q(x)$, where the coefficients of P and Q are given in the ranges *[begin_numer,end_numer)* and *[begin_denom,end_denom)*, respectively. The function is defined over the interval $I = [x, +\infty)$ if *right* is true and $I = (-\infty, x]$ otherwise.
Precondition: Q has no real roots in the interior of I.

template <typename InputIterator>
X_monotone_curve_2 *construct.operator()(InputIterator begin_numer,*
InputIterator end_numer,
InputIterator begin_denom,
InputIterator end_denom,
Algebraic_real_1 lower,
Algebraic_real_1 upper)

Constructs an x-monotone curve supported by the rational function $y = P(x)/Q(x)$, where the coefficients of P and Q are given in the ranges *[begin_numer,end_numer)* and *[begin_denom,end_denom)*, respectively. The function is defined over the interval $I = [lower, upper]$.
Precondition: Q has no real roots in the interior of I.

CGAL::Arr_Bezier_curve_traits_2<RatKernel,AlgKernel,NtTraits>

Class

Definition

The traits class *Arr_Bezier_curve_traits_2<RatKernel,AlgKernel,NtTraits>* is a model of the *ArrangementTraits_2* concept that handles planar Bézier curves. A planar *Bézier curve B* is a parametric curve defined by a sequence of *control points* $p_0, \ldots, p_n$ as follows:

$$B(t) = (X(t), Y(t)) = \sum_{k=0}^{n} p_k \cdot \frac{n!}{k!(n-k)!} \cdot t^k (1-t)^{n-k} \, .$$

where $t \in [0,1]$. The degree of the curve is therefore n — namely, $X(t)$ and $Y(t)$ are polynomials of degree n. Bézier curves have numerous applications in computer graphics and solid modelling. They are used, for example, in free-form sketches and for defining the true-type fonts.

In our representation, we assume that the coordinates of all control points are rational numbers (namely they are given as objects of the *RatKernel::Point_2* type), so both $X(t)$ and $Y(t)$ are polynomials with rational coefficients. The intersection points between curves are however algebraic numbers, and their exact computation is time-consuming. The traits class therefore contains a layer of geometric filtering that performs all computation in an approximate manner whenever possible, and it resorts to exact computations only when the approximate computation fails to produce an unambiguous result.

We therefore require separate representations of the control points and the intersection points. The *NtTraits* should be instantiated with a class that defines nested *Integer*, *Rational* and *Algebraic* number types and supports various operations on them, yielding certified computation results (for example, in can convert rational numbers to algebraic numbers and can compute roots of polynomials with integer coefficients). The other template parameters, *RatKernel* and *AlgKernel* should be geometric kernels templated with the *NtTraits::Rational* and *NtTraits::Algebraic* number types, respectively. It is recommended to instantiate the *CORE_algebraic_number_traits* class as the *NtTraits* parameter, with *Cartesian<NtTraits::Rational>* and *Cartesian<NtTraits::Algebraic>* instantiating the two kernel types, respectively. The number types in this case are provided by the CORE library, with its ability to exactly represent simple algebraic numbers.

While *Arr_Bezier_curve_traits_2<RatKernel,AlgKernel,NtTraits>* models the concept *ArrangementDirectionalXMonotoneTraits_2*, the implementation of the *Arr_mergeable_2* operation does not enforce the input curves to have the same direction as a precondition. Moreover, *Arr_Bezier_curve_traits_2<RatKernel,AlgKernel,NtTraits>* supports the merging of curves of opposite directions.

#include <CGAL/Arr_Bezier_curve_traits_2.h>

Is Model for the Concepts

ArrangementTraits_2
ArrangementDirectionalXMonotoneTraits_2

Types

Arr_Bezier_curve_traits_2<RatKernel,AlgKernel,NtTraits>:: Rational

the *NtTraits::Rational* type (and also the *RatKernel::FT* type).

Arr_Bezier_curve_traits_2<RatKernel,AlgKernel,NtTraits>:: Algebraic

the *NtTraits::Algebraic* type (and also the *AlgKernel::FT* type).

Class Arr_Bezier_curve_traits_2<RatKernel,AlgKernel,NtTraits>::Curve_2

The *Curve_2* class nested within the Bézier traits class is used to represent a Bézier curve of arbitrary degree, which is defined by a sequence of rational control points. In addition to the methods listed below, the I/O operators *operator<<* and *operator>>* for standard output-streams are also supported. The copy constructor and assignment operator are supported as well.

Creation

Arr_Bezier_curve_traits_2<AlgKernel,NtTraits>::Curve_2 B;

default constructor.

template <class InputIterator>
Arr_Bezier_curve_traits_2<AlgKernel,NtTraits>::Curve_2 B(InputIterator pts_begin, InputIterator pts_end);

constructs a Bézier curve as defined by the given range of control points. The value-type of *InputIterator* is *RatKernel::Point_2*.
Precondition: The input range must contain at least two control points.

Access Functions

size_t *B.number_of_control_point() const*

returns the number of control points that define *B*.

typename RatKernel::Point_2

B.control_point(size_t k) const

returns the *k*th control point. Note that the first control point equals the curve source, while the last control point equals its target. The rest of the control points do not lie on the curve.
Precondition: *k* is smaller than the number of control points.

typename RatKernel::Point_2

B(Rational t) returns the point $B(t)$ on the curve that corresponds to the given rational parameter value.

typename AlgKernel::Point_2

B(Algebraic t) returns the point $B(t)$ on the curve that corresponds to the given algebraic parameter value.

Class Arr_Bezier_curve_traits_2<RatKernel,AlgKernel,NtTraits>::Point_2

The *Point_2* class nested within the Bézier traits class is used to represent: (i) an endpoint of a Bézier curve, (ii) a vertical tangency point of a curve, used to subdivide it into x-monotone subcurve, and (iii) an intersection point between two curves. While, points of type (i) have rational coordinates and are given as part of the input, points of the two latter types have algebraic coordinates. However, to speed up the arrangement construction, such point are not computed in an exact manner, and instead are given in an approximate representation. Note that the exact coordinates of a point may only be accessed if it is exactly computed.

In addition to the methods listed below, the copy constructor and assignment operator for *Point_2* objects are also supported.

Creation

Arr_Bezier_curve_traits_2<AlgKernel,NtTraits>::Point_2 p;

default constructor.

Arr_Bezier_curve_traits_2<AlgKernel,NtTraits>::Point_2 p(Curve_2 B, Algebraic t_0);

constructs the point $B(t_0)$ on the given curve. As t_0 is an algebraic number, the point has algebraic coordinates.

Arr_Bezier_curve_traits_2<AlgKernel,NtTraits>::Point_2 p(Curve_2 B, Rational t_0);

constructs the point $B(t_0)$ on the given curve. As t_0 is a rational number, the point has rational coordinates.

Access Functions

std::pair<double, double>

p.approximate() const

returns the approximated coordinates of p.

bool	*p.is_exact() const*	returns whether the coordinates of p are computed in an exact manner.
Algebraic	*p.x() const*	returns the x-coordinate of p. *Precondition*: p is exactly computed.
Algebraic	*p.y() const*	returns the y-coordinate of p. *Precondition*: p is exactly computed.

bool *p.is_rational() const* returns whether the coordinates of p are rational numbers.

typename RatKernel::Point_2

typename RatKernel::Point_2 (p)

casts p to a point with rational coordinates.
Precondition: p has rational coordinates.

Class Arr_Bezier_curve_traits_2<RatKernel,AlgKernel,NtTraits>::X_monotone_curve_2

The *X_monotone_curve_2* class nested within the Bézier traits is used to represent x-monotone subcurves of Bézier curves. The subcurve is defined by a supporting Bézier curve $B(t)$ and a range of definition in the parameter space $[t_1, t_2] \subseteq [0, 1]$, where $B(t_1)$ is the subcurve source and $B(t_2)$ is its target. Note that as the point endpoints may only be approximated, the parameter range defining the subcurve may only be approximately known.

It is not possible to construct x-monotone subcurves directly. Instead, use the *Make_x_monotone_2* functor supplied by the traits class to subdivide a *Curve_2* object into x-monotone subcurves.

Access Functions

Curve_2 *b.supporting_curve() const*

returns the supporting Bézier curve of b.

Point_2 *b.source() const* returns the source point of b.
Point_2 *b.target() const* returns the target point of b.

Point_2 *b.left() const* returns the left (xy-lexicographically smaller) endpoint of b.
Point_2 *b.right() const* returns the right (xy-lexicographically smaller) endpoint of b.

std::pair<double, double>

b.parameter_range() const

return the approximate parameter range defining the subcurve b.

CGAL::Arr_algebraic_segment_traits_2<Coefficient>

Class

Definition

The traits class *Arr_algebraic_segment_traits_2<Coefficient>* is a model of the *ArrangementTraits_2* concept that handles planar algebraic curves of arbitrary degree, and x-monotone of such curves. A planar (real) *algebraic curve* is the vanishing set of a polynomial in two variables, that is, the curve is defined by the defining equation

$$f(x) := \sum_{i+j \leq n} a_{ij} x^i y^j = 0,$$

where n is the degree of the curve.

The traits class allows the construction of algebraic curves, by specifying their implicit equation. x-monotone and vertical segments of a curve can also be defined; unbounded curves and segments are supported. The template parameter *Coefficient* defines the innermost coefficient type of the polynomials. Currently, the types *leda::integer* and *CORE::BigInt* are supported as well as any instance of *CGAL::Sqrt_extension* that is instantiated with one of the integral types above.

#include <CGAL/Arr_algebraic_segment_traits_2.h>

Is Model for the Concepts

ArrangementTraits_2

Types

Arr_algebraic_segment_traits_2<Coefficient>:: enum Site_of_point POINT_IN_INTERIOR = 0, MIN_ENDPOINT = -1, MAX_ENDPOINT = 1 ;

Value to specify whether a point should be in the interior of a segment, or its minimal point, or its maximal point in lexicographic order.

Arr_algebraic_segment_traits_2<Coefficient>:: Polynomial_2

the type for bivariate polynomials, with innermost coefficient type *Coefficient*. Constitutes a model of the concept *Polynomial_d* with two variables. (see page page 261)

Arr_algebraic_segment_traits_2<Coefficient>:: Algebraic_kernel_1

model for the concept *AlgebraicKernel_1*

Arr_algebraic_segment_traits_2<Coefficient>:: Algebraic_real_1

represents coordinates of points. Typedef from *Algebraic_kernel_1::Algebraic_real_1*

Arr_algebraic_segment_traits_2<Coefficient>:: Bound

Typedef from *Algebraic_kernel_1::Bound*

Class Arr_algebraic_segment_traits_2<Coefficient>::Curve_2

Models the *ArrangementTraits_2::Curve_2* concept. Represents algebraic curves. Internally, the type stores topological-geometric information about the particular curve. In order to use internal caching, instances should only be created using the *Construct_curve_2* functor of the traits class.

Polynomial_2	*C.polynomial() const*	returns the defining polynomial of the curve.

Class Arr_algebraic_segment_traits_2<Coefficient>::Point_2

Models the *ArrangementBasicTraits_2::Point_2* concept. Represents points in $\mathbb{R}^2$. Intersection points of algebraic curves are in general non-rational, so we need a data structure that is capable of representing arbitrary points with algebraic coordinates.

The traits class represents algebraic coordinates by the type *Algebraic_real_1*, which is a model of the *AlgebraicReal_1* concept. A point p is stored by a triple $(x, cv, arcno)$, where x is the x-coordinate of a point, cv is an instance of *Curve_2* that contains the point, (and has no vertical line at x), and $arcno$ is an *int*, denoting that p is met as the $arcno$-th point when shooting a vertical ray at x, starting from $-\infty$ (where counting starts with 0).

In addition to the methods listed below, the copy constructor and assignment operator for *Point_2* objects are also supported.

The functor *Construct_point_2* constructs *Point_2* instances.

Algebraic_real_1	*p.x() const*	returns the x-coordinate of p.
Algebraic_real_1	*p.y() const*	returns the y-coordinates of p. **Attention:** As described above, points are not stored by their y-coordinate in *Algebraic_real_1* representation. In fact, this representation must be computed on demand, and might become quite costly for points defined by high-degree polynomials. Therefore, it is recommended to avoid to call this function as much as possible.
Curve_2	*p.curve() const*	returns a *Curve_2* instance that $p$is part of.
int	*p.arcno() const*	returns the arc number of p.
std::pair<double,double>	*p.to_double() const*	returns double-approximations of the x- and y-coordinates.

Class Arr_algebraic_segment_traits_2<Coefficient>::X_monotone_curve_2

Models the *ArrangementBasicTraits_2::X_monotone_curve_2* concept. Represents terminal segments of an algebraic curves, that means vertical segments or x-monotone segments with no critical x-coordinate in the interior of their x-range. Terminal segments might either be bounded or unbounded. By definition, each interior point of a non-vertical segment has the same arc number (see the documentation of type *Point_2* above, which is called the *arc number* of the segment (note the arc number at the endpoints might differ). Such segments are represented internally by a 4-tuple $(p,q,cv,arcno)$, where p and q are the endpoints, cv is the *supporting curve* that the segment belongs to, and arcno is the arc number of the segment.

Arbitrary (weakly) x-monotone segments are presented by a range of *X_monotone_curve_2* instances, whose union equals the segment. The functor *Construct_x_monotone_segment_2* allows their construction. To construct all (maximal) terminal segments of a curve, use the *Make_x_monotone_2* functor supplied by the traits class.

Curve_2	*s.curve() const*	returns the supporting algebraic curve of s.
bool	*s.is_vertical() const*	returns whether s is a vertical segment.
bool	*s.is_finite(CGAL::Arr_curve_end ce) const*	
		returns whether s has a finite endpoint on the left (if *ce==CGAL::ARR_MIN_END*) or on the right (if *ce==CGAL::ARR_MAX_END*).
Point_2	*s.curve_end(CGAL::Arr_curve_end ce) const*	
		returns the left or right endpoint of $s$for *ce==CGAL::ARR_MIN_END* and *ce==CGAL::ARR_MAX_END*. *Precondition*: (The corresponding curve end is finite)
int	*s.arcno() const*	returns the arc number of the segment. *Precondition*: (The segment is non-vertical)
Algebraic_real_1	*s.x() const*	returns the x-coordinate of a vertical segment. *Precondition*: (The segment is vertical)

Object Creation Functors

Curves, points, and x-monotone segments are created by special functors. The functors are not default constructible; the only possibility to obtain them is by the corresponding accessing functions.

Class Arr_algebraic_segment_traits_2<Coefficient>::Construct_curve_2

Curve_2	*fo(Polynomial_2 p)*	Returns a *Curve_2* object that represents the curve defined by the polynomial p

Curve_2	*fo(std::string s)*	Returns a *Curve_2* object specified by *s*. The passed string represents the defining polynomial of the curve and must be given in a MAPLE-readable format using "x" as first and "y" as second variable, e.g., "(x^3*y-2*x)*(-6*x-y^3*x^6)" for integer coefficients, and "3/2*x*y^4-5/7*x^2+3/1" for rational coefficients.

Class Arr_algebraic_segment_traits_2<Coefficient>::Construct_point_2

Point_2 *fo(Algebraic_real_1 x, Curve_2 cv, int arcno)*

Returns a *Point_2* object that represents the *arcno*-th point in the fiber of *cv* at *x*-coordinate *x*, counted from the bottom, starting with zero.
Precondition: (*cv* must not have a vertical line at *x*, and $0 \leq$ *arcno* $< c$, where c is the number of points in the fiber of *cv* at *x*.)

Point_2 *fo(Algebraic_real_1 x, X_monotone_curve_2 xcv)*

Returns a *Point_2* object that represents the point on *xcv* at *x*-coordinate *x*
Precondition: (*x* is in the *x*-range of *xcv*.)

Point_2 *fo(Algebraic_real_1 x, Algebraic_real_1 y)*

Returns a *Point_2* object that represents (x,y)

Point_2 *fo(Coefficient x, Coefficient y)*

Returns a *Point_2* object that represents (x,y)

Point_2	*fo(Bound x, Bound y)*	Returns a *Point_2* object that represents (x,y)
Point_2	*fo(int x, int y)*	Returns a *Point_2* object that represents (x,y)

Class Arr_algebraic_segment_traits_2<Coefficient>::Construct_x_monotone_segment_2

template<class OutputIterator>
OutputIterator *fo(Curve_2 cv, Point_2 end_min, Point_2 end_max, OutputIterator out)*

Writes a sequence of *X_monotone_curve_2* objects (terminal segments) into *out*. These terminal segments compose an *x*-monotone (or vertical) segment of the curve *cv* that starts in *end_min*, and ends in *end_max*.
Precondition: (*end_min* must have a unique *x*-monotone segment to its right, or *end_max* must have a unique *x*-monotone segment to its left. Furthermore, *end_min* and *end_max* must be connected by an *x*-monotone segment of *cv*)

template<class OutputIterator>
OutputIterator *fo(Curve_2 cv, Point_2 p, Site_of_point site_of_p, OutputIterator out)*

Writes a sequence of *X_monotone_curve_2* objects into *out*. These segments form an *x*-monotone (or vertical) segment of the curve *cv*.
If *site_of_p==POINT_IN_INTERIOR*, the maximal segment is returned that contains *p* in its interior.
If *site_of_p==MIN_ENDPOINT*, the maximal segment is returned that contains *p* as its left endpoint.
If *site_of_p==MAX_ENDPOINT*, the maximal segment is returned that contains *p* as its left endpoint.
Precondition: (If *site_of_p==POINT_IN_INTERIOR*, *p* must be an interior point of an *x*-monotone or a vertical segment. If *site_of_p==MIN_ENDPOINT*, *p* must either have a unique *x*-monotone segment to the right, or a vertical segment from *p* upwards. If *site_of_p==MAX_ENDPOINT*, *p* must either have a unique *x*-monotone segment to the left, or a vertical segment from *p* downwards.)

template<class OutputIterator>
OutputIterator *fo(Point_2 p, Point_2 q, OutputIterator out)*

Writes a sequence of *X_monotone_curve_2* objects into *out*. These segments form a straight-line segment connecting the points *p* and *q*. If *p* and *q* share the same *x*-coordinate, the constructed vertical segment consists of only one *X_monotone_curve_2* object and can be computed efficiently. In the non-vertical case, the construction is only possible if *p* and *q* have both rational x- and y-coordinates.
Precondition: (*p* must not be equal to *q*.)

Accessing functor objects

Construct_curve_2 *traits.construct_curve_2_object() const*
Construct_point_2 *traits.construct_point_2_object() const*
Construct_x_monotone_segment_2

traits.construct_x_monotone_segment_2_object() const

Class

CGAL::Arr_curve_data_traits_2<Tr,XData,Mrg,CData,Cnv>

Definition

The class *Arr_curve_data_traits_2<Tr,XData,Mrg,CData,Cnv>* is a model of the *ArrangementTraits_2* concept and serves as a decorator class that allows the extension of the curves defined by the base traits-class (the *Tr* parameter), which serves as a geometric traits-class (a model of the *ArrangementTraits_2* concept), with extraneous (non-geometric) data fields.

The traits class inherits its point type from *Traits::Point_2*, and defines an extended *Curve_2* and *X_monotone_curve_2* types, as detailed below.

Each *Curve_2* object is associated with a single data field of type *CData*, and each *X_monotone_curve_2* object is associated with a single data field of type *XData*. When a curve is subdivided into x-monotone subcurves, its data field is converted using the conversion functor, which is specified by the *Cnv* template-parameter, and the resulting objects is copied to all *X_monotone_curve_2* objects induced by this curve. The conversion functor should provide an operator with the following prototype:
XData operator() (const CData& d) const;

By default, the two data types are the same, so the conversion operator is trivial:

CData = *XData*
Cnv = *_Default_convert_functor<CData,XData>*

In case two (or more) x-monotone curves overlap, their data fields are merged to a single field, using the merge functor functor, which is specified by the *Mrg* template-parameter. This functor should provide an operator with the following prototype:
XData operator() (const XData& d1, const XData& d2) const;
which returns a single data object that represents the merged data field of *d1* and *d2*. The x-monotone curve that represents the overlap is associated with the output of this functor.

#include <CGAL/Arr_curve_data_traits_2.h>

Is Model for the Concepts

ArrangementTraits_2

Types

typedef Tr	*Base_traits_2;*	the base traits-class.
typedef typename Base_traits_2::Curve_2	*Base_curve_2;*	the base curve.
typedef typename Base_traits_2::X_monotone_curve_2	*Base_x_monotone_curve_2;*	the base x-monotone curve curve.

typedef Mrg	*Merge;*	the merge functor.
typedef Cnv	*Convert;*	the conversion functor.

typedef CData	*Curve_data;*	the type of data associated with curves.
typedef XData	*X_monotone_curve_data;*	the type of data associated with x-monotone curves.

Inherits From

Base_traits_2

Class Arr_curve_data_traits_2<Tr,XData,Mrg,CData,Cnv>::Curve_2

The *Curve_2* class nested within the curve-data traits extends the *Base_traits_2::Curve_2* type with an extra data field of type *Data*.

Inherits From

Base_curve_2

Creation

Arr_curve_data_traits_2<Tr,XData,Mrg,CData,Cnv>::Curve_2 cv;

default constructor.

Arr_curve_data_traits_2<Tr,XData,Mrg,CData,Cnv>::Curve_2 cv(Base_curve_2 base);

constructs curve from the given *base* curve with uninitialized data field.

Arr_curve_data_traits_2<Tr,XData,Mrg,CData,Cnv>::Curve_2 cv(Base_curve_2 base, Data data);

constructs curve from the given *base* curve with an attached *data* field.

Access Functions

Curve_data	*cv.data() const*	returns the data field (a non-const version, which returns a reference to the data object, is also available).
void	*cv.set_data(Curve_data data)*	sets the data field.

Class Arr_curve_data_traits_2<Tr,XData,Mrg,CData,Cnv>::X_monotone_curve_2

The *X_monotone_curve_2* class nested within the curve-data traits extends the *Base_traits_2::X_monotone_curve_2* type with an extra data field.

Inherits From

Base_x_monotone_curve_2

Creation

Arr_curve_data_traits_2<Tr,XData,Mrg,CData,Cnv>::X_monotone_curve_2 xcv;

default constructor.

Arr_curve_data_traits_2<Tr,XData,Mrg,CData,Cnv>::X_monotone_curve_2 xcv(Base_x_monotone_curve_2 base)

constructs an *x*-monotone curve from the given *base* curve with uninitialized data field.

Arr_curve_data_traits_2<Tr,XData,Mrg,CData,Cnv>::X_monotone_curve_2 xcv(Base_x_monotone_curve_2 base, X_monotone_curve_data data)

constructs an *x*-monotone curve from the given *base* *x*-monotone curve with an attached *data* field.

Access Functions

X_monotone_curve_data *xcv.data() const*

returns the field (a non-const version, which returns a reference to the data object, is also available).

void *xcv.set_data(X_monotone_curve_data data)*

sets the data field.

CGAL::Arr_consolidated_curve_data_traits_2<Traits,Data>

Class

Definition

The class *Arr_consolidated_curve_data_traits_2<Traits,Data>* is a model of the concept *ArrangementTraits_2*, and serves as a decorator class that enables the extension of the curve type defined by the *Traits* parameter. The traits class inherits its point type from *Traits::Point_2*, and defines the types *Curve_2* and *X_monotone_curve_2* extended with extraneous data fields of type *Data*.

Each *Curve_2* object is associated with a single data field of type *Data*, and each *X_monotone_curve_2* object is associated with a set of unique data objects. When a curve is subdivided into x-monotone subcurves, all resulting subcurves are associated with a list containing a single data object, copied from the inducing curve. When an x-monotone curve is split, its data set is duplicated, and inserted into the sets of both resulting subcurves. In case two (or more) x-monotone curves overlap, their data sets are consolidated, and are inserted into the set of the x-monotone curve that represents the overlap.

#include <CGAL/Arr_consolidated_curve_data_traits_2.h>

Is Model for the Concepts

ArrangementTraits_2

Inherits From

```
Arr_curve_data_traits_2<Traits,
                        _Unique_list<Data>,
                        _Consolidate_unique_lists<Data>,
                        Data>
```

Types

typedef Traits *Base_traits_2;* the base traits-class.

typedef typename Base_traits_2::Curve_2
Base_curve_2; the base curve.

typedef typename Base_traits_2::X_monotone_curve_2
Base_x_monotone_curve_2;
the base x-monotone curve curve.

Arr_consolidated_curve_data_traits_2<Traits,Data>:: typedef Data_container

a set of data objects that is associated with an x-monotone curve.

Arr_consolidated_curve_data_traits_2<Traits,Data>:: typedef Data_iterator

a non-mutable iterator for the data objects in the data container.

Class Arr_consolidated_curve_data_traits_2<Traits,Data>::Data_container

The *Data_container* class nested within the consolidated curve-data traits and associated with the *Traits::X_monotone_curve_2* type is maintained as a list with unique data objects. This representation is simple and efficient in terms of memory consumption. It also requires that the *Data* class supports only the equality operator. Note however that most set operations require linear time.

Creation

Arr_consolidated_curve_data_traits_2<Traits,Data>:: Data_container dset;

default constructor.

Arr_consolidated_curve_data_traits_2<Traits,Data>:: Data_container dset(Data data);

constructs set containing a single *data* object.

Access Functions

std::size_t	*dset.size() const*	returns the number of data objects in the set.
Data_iterator	*dset.begin() const*	returns an iterator pointing to the first data object.
Data_iterator	*dset.end() const*	returns a past-the-end iterator for the data objects.
Data	*dset.front() const*	returns the first data object inserted into the set. *Precondition*: The number of data objects is not 0.
Data	*dset.back() const*	returns the last data object inserted into the set. *Precondition*: The number of data objects is not 0.

Predicates

bool *dset == Data_container other*

check if the two sets contain the same data objects (regardless of order).

Data_iterator *dset.find(Data data)* find the given *data* object in the set and returns an iterator for this object, or *end()* if it is not found.

Modifiers

bool *dset.insert(Data data)*

inserts the given *data* object into the set. Returns *true* on success, or *false* if the set already contains the object.

bool	*dset.erase(Data data)*	
		erases the given *data* object from the set. Returns *true* on success, or *false* if the set does not contain the object.
void	*dset.clear()*	clears the set.

Concept

ArrangementInputFormatter

A model for the ArrangementInputFormatter concept supports a set of functions that enable reading an arrangement from an input stream using a specific format.

Types

ArrangementInputFormatter:: Arrangement_2 — the type of arrangement to input.

typedef typename Arrangement_2::Point_2
Point_2; — the point type.

typedef typename Arrangement_2::X_monotone_curve_2
X_monotone_curve_2; — the x-monotone curve type.

typedef typename Arrangement_2::Size
Size;

typedef typename Arrangement_2::Vertex_handle
Vertex_handle;

typedef typename Arrangement_2::Halfedge_handle
Halfedge_handle;

typedef typename Arrangement_2::Face_handle
Face_handle;

Creation

ArrangementInputFormatter inf; — default constructor.

ArrangementInputFormatter inf(std::istream& is);
constructs a formatter that reads from *is*.

void *inf.set_in(std::istream& is)*
directs *inf* to read from *is*.

Access Functions

std::istream& *inf.in()* — returns the stream that *inf* reads from.
Precondition: *inf* is directed to a valid output stream.

Formatted Input Functions

void	*inf.read_arrangement_begin()*	
		reads a message indicating the beginning of the arrangement.
void	*inf.read_arrangement_end()*	
		reads a message indicating the end of the arrangement.
Size	*inf.read_size(const char *label = NULL)*	
		reads a size value, which is supposed to be preceded by the given label.
void	*inf.read_vertices_begin()*	
		reads a message indicating the beginning of the vertex records.
void	*inf.read_vertices_end()*	
		reads a message indicating the end of the vertex records.
void	*inf.read_edges_begin()*	
		reads a message indicating the beginning of the edge records.
void	*inf.read_edges_end()*	reads a message indicating the end of the edge records.
void	*inf.read_faces_begin()*	
		reads a message indicating the beginning of the face records.
void	*inf.read_faces_end()*	reads a message indicating the end of the face records.
void	*inf.read_vertex_begin()*	
		reads a message indicating the beginning of a single vertex record.
void	*inf.read_vertex_end()*	reads a message indicating the end of a single vertex record.
std::size_t	*inf.read_vertex_index()*	
		reads and returns a vertex index.
void	*inf.read_point(Point_2& p)*	
		reads a point.

void	*inf.read_vertex_data(Vertex_handle v)*	
		reads an auxiliary vertex-data object and associates it with the vertex *v*.
void	*inf.read_edge_begin()*	
		reads a message indicating the beginning of a single edge record.
void	*inf.read_edge_end()*	reads a message indicating the end of a single edge record.
std::size_t	*inf.read_halfedge_index()*	
		reads and returns halfedge index.
void	*inf.read_x_monotone_curve(X_monotone_curve_2& c)*	
		reads an *x*-monotone curve.
void	*inf.read_halfegde_data(Halfedge_handle he)*	
		reads an auxiliary halfedge-data object and associates it with the halfedge *he*.
void	*inf.read_face_begin()*	reads a message indicating the beginning of a single face record.
void	*inf.read_face_end()*	reads a message indicating the end of a single face record.
void	*inf.read_outer_ccb_begin()*	
		reads a message indicating the beginning of the outer CCB of the current face.
void	*inf.read_outer_ccb_end()*	
		reads a message indicating the end of the outer CCB of the current face.
void	*inf.read_holes_begin()*	
		reads a message indicating the beginning of the container of holes inside the current face.
void	*inf.read_holes_end()*	reads a message indicating the end of the container of holes inside the current face.
void	*inf.read_inner_ccb_begin()*	
		reads a message indicating the beginning of an inner CCB of the current face.

void *inf.read_inner_ccb_end()*

reads a message indicating the end of an inner CCB of the current face.

void *inf.read_ccb_halfedges_begin()*

reads a message indicating the beginning a connected component boundary.

void *inf.read_ccb_halfedges_end()*

reads a message indicating the end of a connected component boundary.

void *inf.read_isolated_vertices_begin()*

reads a message indicating the beginning of the container of isolated vertices inside the current face.

void *inf.read_isolated_vertices_end()*

reads a message indicating the end of the container of isolated vertices inside the current face.

void *inf.read_face_data(Face_handle f)*

reads an auxiliary face-data object and associates it with the face f.

Has Models

Arr_text_formatter<Arrangement> (page 2338)
Arr_face_extended_text_formatter<Arrangement> (page 2339)
Arr_extended_dcel_text_formatter<Arrangement> (page 2340)

Concept

ArrangementOutputFormatter

A model for the ArrangementOutputFormatter concept supports a set of functions that enable writing an arrangement to an output stream using a specific format.

Types

ArrangementOutputFormatter:: Arrangement_2

the type of arrangement to output.

typedef typename Arrangement_2::Point_2

Point_2; the point type.

typedef typename Arrangement_2::X_monotone_curve_2

X_monotone_curve_2;

the x-monotone curve type.

typedef typename Arrangement_2::Size

Size;

typedef typename Arrangement_2::Vertex_const_handle

Vertex_const_handle;

typedef typename Arrangement_2::Halfedge_const_handle

Halfedge_const_handle;

typedef typename Arrangement_2::Face_const_handle

Face_const_handle;

Creation

ArrangementOutputFormatter outf; default constructor.

ArrangementOutputFormatter outf(std::ostream& os);

constructs a formatter that writes to *os*.

void *outf.set_out(std::ostream& os)*

directs *outf* to write to *os*.

Access Functions

std::ostream&	*outf.out()*	returns the stream that *outf* writes to. *Precondition*: *outf* is directed to a valid output stream.

Formatted Output Functions

void	*outf.write_arrangement_begin()*	
		writes a message indicating the beginning of the arrangement.
void	*outf.write_arrangement_end()*	
		writes a message indicating the end of the arrangement.
void	*outf.write_size(const char *label, Size size)*	
		writes a size value, preceded by a given label.
void	*outf.write_vertices_begin()*	
		writes a message indicating the beginning of the vertex records.
void	*outf.write_vertices_end()*	
		writes a message indicating the end of the vertex records.
void	*outf.write_edges_begin()*	
		writes a message indicating the beginning of the edge records.
void	*outf.write_edges_end()*	
		writes a message indicating the end of the edge records.
void	*outf.write_faces_begin()*	
		writes a message indicating the beginning of the face records.
void	*outf.write_faces_end()*	
		writes a message indicating the end of the face records.
void	*outf.write_vertex_begin()*	
		writes a message indicating the beginning of a single vertex record.
void	*outf.write_vertex_end()*	
		writes a message indicating the end of a single vertex record.

void *outf.write_vertex_index(std::size_t idx)*

writes a vertex index.

void *outf.write_point(Point_2 p)*

writes a point.

void *outf.write_vertex_data(Vertex_const_handle v)*

writes the auxiliary data associated with the vertex.

void *outf.write_edge_begin()*

writes a message indicating the beginning of a single edge record.

void *outf.write_edge_end()*

writes a message indicating the end of a single edge record.

void *outf.write_halfedge_index(std::size_t idx)*

writes a halfedge index.

void *outf.write_x_monotone_curve(X_monotone_curve_2 c)*

writes an *x*-monotone curve.

void *outf.write_halfegde_data(Halfedge_const_handle he)*

writes the auxiliary data associated with the halfedge.

void *outf.write_face_begin()*

writes a message indicating the beginning of a single face record.

void *outf.write_face_end()* writes a message indicating the end of a single face record.

void *outf.write_outer_ccb_begin()*

writes a message indicating the beginning of the outer CCB of the current face.

void *outf.write_outer_ccb_end()*

writes a message indicating the end of the outer CCB of the current face.

void *outf.write_holes_begin()*

writes a message indicating the beginning of the container of holes inside the current face.

void *outf.write_holes_end()*

writes a message indicating the end of the container of holes inside the current face.

void *outf.write_ccb_halfedges_begin()*

writes a message indicating the beginning a connected component's boundary.

void *outf.write_ccb_halfedges_end()*

writes a message indicating the end of a connected component's boundary.

void *outf.write_isolated_vertices_begin()*

writes a message indicating the beginning of the container of isolated vertices inside the current face.

void *outf.write_isolated_vertices_end()*

writes a message indicating the end of the container of isolated vertices inside the current face.

void *outf.write_face_data(Face_const_handle f)*

writes the auxiliary data associated with the face.

Has Models

Arr_text_formatter<Arrangement> (page 2338)
Arr_face_extended_text_formatter<Arrangement> (page 2339)
Arr_extended_dcel_text_formatter<Arrangement> (page 2340)

Class

CGAL::Arr_text_formatter<Arrangement>

Definition

Arr_text_formatter<Arrangement> defines the format of an arrangement in an input or output stream (typically a file stream), thus enabling reading and writing an *Arrangement* instance using a simple text format. The arrangement is assumed to store no auxiliary data with its DCEL records (and if there are such records they will not be written or read by the formatter).

The *Arr_text_formatter<Arrangement>* class assumes that the nested *Point_2* and the *Curve_2* types defined by the *Arrangement* template-parameter can both be written to an input stream using the << operator and read from an input stream using the >> operator.

#include <CGAL/IO/Arr_text_formatter.h>

Is Model for the Concepts

ArrangementInputFormatter
ArrangementOutputFormatter

See Also

read (page 2228)
write (page 2230)

CGAL::Arr_face_extended_text_formatter<Arrangement>

Class

Definition

Arr_face_extended_text_formatter<Arrangement> defines the format of an arrangement in an input or output stream (typically a file stream), thus enabling reading and writing an *Arrangement* instance using a simple text format. The *Arrangement* class should be instantiated with a DCEL class which in turn instantiates the *Arr_face_extended_dcel* template with a *FaceData* type. The formatter supports reading and writing the data objects attached to the arrangement faces as well.

The *Arr_face_extended_text_formatter<Arrangement>* class assumes that the nested *Point_2* and the *Curve_2* types defined by the *Arrangement* template-parameter and that the *FaceData* type can all be written to an input stream using the << operator and read from an input stream using the >> operator.

#include <CGAL/IO/Arr_text_formatter.h>

Is Model for the Concepts

ArrangementInputFormatter
ArrangementOutputFormatter

See Also

read (page 2228)
write (page 2230)
Arr_face_extended_dcel<Traits,FData,V,H,F> (page 2248)

Class

CGAL::Arr_extended_dcel_text_formatter<Arrangement>

Definition

Arr_extended_dcel_text_formatter<Arrangement> defines the format of an arrangement in an input or output stream (typically a file stream), thus enabling reading and writing an *Arrangement* instance using a simple text format. The *Arrangement* class should be instantiated with a DCEL class which in turn instantiates the *Arr_extended_dcel* template with the *VertexData*, *HalfedgeData* and *FaceData* types. The formatter supports reading and writing the data objects attached to the arrangement vertices, halfedges and faces.

The *Arr_extended_dcel_text_formatter<Arrangement>* class assumes that the nested *Point_2* and the *Curve_2* types defined by the *Arrangement* template-parameter, as well as the *VertexData*, *HalfedgeData* and *FaceData* types, can all be written to an input stream using the << operator and read from an input stream using the >> operator.

#include <CGAL/IO/Arr_text_formatter.h>

Is Model for the Concepts

ArrangementInputFormatter
ArrangementOutputFormatter

See Also

read (page 2228)
write (page 2230)
Arr_extended_dcel<Traits,VData,HData,FData,V,H,F> (page 2249)

ArrangementPointLocation_2

Concept

Definition

A model of the ArrangementPointLocation_2 concept can be attached to an *Arrangement_2* instance and answer point-location queries on this arrangement. Namely, given a *Arrangement_2::Point_2* object, representing a point in the plane, it returns the arrangement cell containing it. In the general case, the query point is contained inside an arrangement face, but in degenerate situations it may lie on an edge or coincide with an arrangement vertex.

Types

ArrangementPointLocation_2:: Arrangement_2

the associated arrangement type.

ArrangementPointLocation_2:: Point_2 — equivalent to *Arrangement_2::Point_2*.

Creation

ArrangementPointLocation_2 pl; — default constructor.

ArrangementPointLocation_2 pl(Arrangement_2 arr);

constructs a point-location object *pl* attached to the given arrangement *arr*.

Query Functions

Object pl.locate(Point_2 q) const

locates the arrangement cell that contains the query point *q* and returns a handle for this cell. The function returns an *Object* instance that wraps either of the following types:

- *Arrangement_2::Face_const_handle*, in case *q* is contained inside an arrangement face;
- *Arrangement_2::Halfedge_const_handle*, in case *q* lies on an arrangement edge;
- *Arrangement_2::Vertex_const_handle*, in case *q* coincides with an arrangement vertex.

Precondition: *pl* is attached to a valid arrangement instance.

Operations

void *pl.attach(Arrangement_2 arr)*

attaches *pl* to the given arrangement *arr*.

void *pl.detach()*

detaches *pl* from the arrangement it is currently attached to.

Has Models

Arr_naive_point_location<Arrangement>
Arr_walk_along_a_line_point_location<Arrangement>
Arr_trapezoid_ric_point_location<Arrangement>
Arr_landmarks_point_location<Arrangement,Generator>

ArrangementVerticalRayShoot_2

Definition

A model of the ArrangementVerticalRayShoot_2 concept can be attached to an *Arrangement_2* instance and answer vertical ray-shooting queries on this arrangement. Namely, given a *Arrangement_2::Point_2* object, representing a point in the plane, it returns the arrangement feature (edge or vertex) that lies strictly above it (or below it). By "strictly" we mean that if the query point lies on an arrangement edge (or on an arrangement vertex) this edge will *not* be the query result, but the feature lying above or below it. (An exception to this rule is the degenerate situation where the query point lies in the interior of a vertical edge.) Note that it may happen that the query point lies above the upper envelope (or below the lower envelope) of the arrangement, so that the vertical ray emanating from it may go to infinity without hitting any arrangement feature on its way. In this case the unbounded face is returned.

Types

ArrangementVerticalRayShoot_2:: Arrangement_2

the associated arrangement type.

ArrangementVerticalRayShoot_2:: Point_2 — equivalent to *Arrangement_2::Point_2*.

Creation

ArrangementVerticalRayShoot_2 rs; — default constructor.

ArrangementVerticalRayShoot_2 rs(Arrangement_2 arr);

constructs a ray-shooting object *rs* attached to the given arrangement *arr*.

Query Functions

Object *rs.ray_shoot_up(Point_2 q) const*

locates the arrangement feature that is first hit by an upward-directed vertical ray emanating from the query point q, and returns a handle for this feature. The function returns an *Object* instance that is a wrapper for one of the following types:

- *Arrangement_2::Halfedge_const_handle*, in case the vertical ray hits an arrangement edge;
- *Arrangement_2::Vertex_const_handle*, in case the vertical ray hits an arrangement vertex.
- *Arrangement_2::Face_const_handle* for the unbounded arrangement face, in case q lies above the upper envelope of the arrangement.

Precondition: *rs* is attached to a valid arrangement instance.

Object *rs.ray_shoot_down(Point_2 q) const*

locates the arrangement feature that is first hit by a downward-directed vertical ray emanating from the query point *q*, and returns a handle for this feature. The function returns an *Object* instance that is a wrapper for one of the following types:

- *Arrangement_2::Halfedge_const_handle*, in case the vertical ray hits an arrangement edge;
- *Arrangement_2::Vertex_const_handle*, in case the vertical ray hits an arrangement vertex.
- *Arrangement_2::Face_const_handle* for the unbounded arrangement face, in case *q* lies below the lower envelope of the arrangement.

Precondition: *rs* is attached to a valid arrangement instance.

Operations

void *rs.attach(Arrangement_2 arr)*

attaches *rs* to the given arrangement *arr*.

void *rs.detach()* detaches *rs* from the arrangement it is currently attached to.

Has Models

Arr_naive_point_location<Arrangement>
Arr_walk_along_a_line_point_location<Arrangement>
Arr_trapezoid_ric_point_location<Arrangement>
Arr_landmarks_point_location<Arrangement,Generator>

CGAL::Arr_naive_point_location<Arrangement>

Class

Definition

The *Arr_naive_point_location<Arrangement>* class implements a naive algorithm that traverses all the vertices and halfedges in the arrangement in search for an answer to a point-location query. The query time is therefore linear in the complexity of the arrangement. Naturally, this point-location strategy could turn into a heavy time-consuming process when applied to dense arrangements.

#include <CGAL/Arr_naive_point_location.h>

Is Model for the Concepts

ArrangementPointLocation_2
ArrangementVerticalRayShoot_2

Class

CGAL::Arr_walk_along_line_point_location<Arrangement>

Definition

The *Arr_walk_along_line_point_location<Arrangement>* class implements a very simple point-location (and vertical ray-shooting) strategy that improves the naive one. The algorithm considers an imaginary vertical ray emanating from the query point, and simulates a walk along the zone of this ray, starting from the unbounded face until reaching the query point. In dense arrangements this walk can considerably reduce the number of traversed arrangement edges, with respect to the naïve algorithm.

The walk-along-a-line point-location object (just like the naïve one) does not use any auxiliary data structures. Thus, attaching it to an existing arrangement takes constant time, and any ongoing updates to this arrangement do not affect the point-location object. It is therefore recommended to use the "walk" point-location strategy for arrangements that are constantly changing, especially if the number of issued queries is not large.

#include <CGAL/Arr_walk_along_line_point_location.h>

Is Model for the Concepts

ArrangementPointLocation_2
ArrangementVerticalRayShoot_2

CGAL::Arr_trapezoid_ric_point_location<Arrangement>

Class

Definition

The *Arr_trapezoid_ric_point_location<Arrangement>* class implements the incremental randomized algorithm introduced by Mulmuley [Mul90] as presented by Seidel [Sei91] (see also [dBvKOS00, Chapter 6]). It subdivides each arrangement face to pseudo-trapezoidal cells, each of constant complexity, and constructs and maintains a linear-size search structure on top of these cells, such that each query can be answered in $O(\log n)$ time, where n is the complexity of the arrangement.

Constructing the search structures takes $O(n \log n)$ expected time and may require a small number of rebuilds [HKH12]. Therefore attaching a trapezoidal point-location object to an existing arrangement may incur some overhead in running times. In addition, the point-location object needs to keep its auxiliary data structures up-to-date as the arrangement goes through structural changes. It is therefore recommended to use this point-location strategy for static arrangements (or arrangement that do not alter frequently), and when the number of issued queries is relatively large.

This strategy supports arbitrary subdivisions, including unbounded ones.

#include <CGAL/Arr_trapezoid_ric_point_location.h>

Is Model for the Concepts

ArrangementPointLocation_2
ArrangementVerticalRayShoot_2

Creation

Arr_trapezoid_ric_point_location<Arrangement> pl(bool with_guarantees = true);

If with_guarantees is set to true, the cunstruction performs rebuilds in order to guarantee a resulting structure with linear size and logarithmic query time. Otherwise the structure has expected linear size and expected logarithmic query time.

Arr_trapezoid_ric_point_location<Arrangement> pl(Arrangement arr, bool with_guarantees = true);

Constructs a point location search structure for the given arrangement. If with_guarantees is set to true, the cunstruction performs rebuilds in order to guarantee a resulting structure with linear size and logarithmic query time. Otherwise the structure has expected linear size and expected logarithmic query time.

Modifiers

void *pl.with_guarantees(bool with_guarantees)*

If with_guarantees is set to true, the structure will guarantee linear size and logarithmic query time, that is, this function may cause a reconstruction of the data structure.

CGAL::Arr_landmarks_point_location<Arrangement,Generator>

The *Arr_landmarks_point_location<Arrangement,Generator>* class implements a Jump & Walk algorithm, where special points, referred to as "landmarks", are chosen in a preprocessing stage, their place in the arrangement is found, and they are inserted into a data-structure that enables efficient nearest-neighbor search (a KD-tree). Given a query point, the nearest landmark is located and a "walk" strategy is applied from the landmark to the query point.

There are various strategies to select the landmark set in the arrangement, where the strategy is determined by the *Generator* template parameter. The following landmark-generator classes are available:

Arr_landmarks_vertices_generator — The arrangement vertices are used as the landmarks set.

Arr_random_landmarks_generator — n random points in the bounding box of the arrangement are chosen as the landmarks set.

Arr_halton_landmarks_generator — n Halton points in the bounding box of the arrangement are chosen as the landmarks set.

Arr_middle_edges_landmarks_generator — The midpoint of each arrangement edge is computed, and the resulting set of points is used as the landmarks set. This generator can be applied only for arrangements of line segments.

Arr_grid_landmarks_generator — A set of n landmarks are chosen on a $\lceil\sqrt{n}\rceil \times \lceil\sqrt{n}\rceil$ grid that covers the bounding box of the arrangement.

The *Arr_landmarks_vertices_generator* class is the default generator and used if no *Generator* parameter is specified.

It is recommended to use the landmarks point-location strategy when the application frequently issues point-location queries on a rather static arrangement that the changes applied to it are mainly insertions of curves and not deletions of them.

#include <CGAL/Arr_landmarks_point_location.h>

Is Model for the Concepts

ArrangementPointLocation_2
ArrangementVerticalRayShoot_2

Function

CGAL::locate

Definition

The function *locate* performs a batched point-location operation on a given arrangement. It accepts a range of query points, and locates each point in the arrangement. The query results are returned through the output iterator. Each result is given as a pair of the query point and an object representing the arrangement feature that contains it, namely an *Object* that may be either *Face_const_handle*, *Halfedge_const_handle*, or *Vertex_const_hanlde*. The resulting pairs in the output sequence are sorted in increasing *xy*-lexicographical order of the query points. The function returns a past-the-end iterator of the output sequence.

#include <CGAL/Arr_batched_point_location.h>

```
template<typename Traits, typename Dcel, typename PointsIterator, typename OutputIterator>
OutputIterator          locate( Arrangement_2<Traits,Dcel> arr,
                                PointsIterator points_begin,
                                PointsIterator points_end,
                                OutputIterator oi)
```

Requirements

- *InputIterator::value_type* must be *Traits::Point_2*.
- *OutputIterator::value_type* must be *std::pair<Traits::Point_2,Object>*.

CGAL::decompose

Definition

The function *decompose* produces the symbolic vertical decomposition of a given arrangement, performing a batched vertical ray-shooting query from all arrangement vertices, such that every vertex is associated with a pair of objects, one corresponds to the arrangement feature that lies below it, and the other corresponds to the feature that lies above it. The output of this function can be readily used for inserting vertical walls and physically decomposing the arrangement into pseudo-trapezoids. To do this, it is convenient to process the vertices in an ascending *xy*-lexicographic order. The visible objects are therefore returned through an output iterator, which pairs each finite arrangement vertex with the two features it "sees", such that the vertices are given in ascending *xy*-lexicographic order.

#include <CGAL/Arr_vertical_decomposition_2.h>

template<typename Traits, typename Dcel, typename OutputIterator>
OutputIterator decompose(Arrangement_2<Traits,Dcel> arr, OutputIterator oi)

Produces the symbolic vertical decomposition of the *arr* arrangement. More precisely, it performs a batched vertical ray-shooting query from all arrangement vertices, such that every vertex is associated with a pair of objects, one corresponding to the arrangement feature that lies below it, while the other corresponds to the feature that lies above it. The query results are returned through the output iterator, which pairs each finite arrangement vertex with a pair of *Object*s, the first represents the feature below the vertex, and the second represents the feature that lies above it. Each *Object* may be one of the following:

- *Halfedge_const_handle*, if the vertex is located above (or below) an edge. The given halfedge is always directed from right to left. In case there is no concrete edge below (or above) the vertex, and the arrangement is unbounded, then the object returned is a *fictitious* halfedge.
- *Face_const_handle*, in case there is no edge below (or above) the vertex, and the arrangement is bounded.
- *Vertex_const_handle*, in case the vertex is located vertically above (or below) another arrangement vertex.
- An empty object, in case the vertex is the top end-vertex of a vertical edge, we define there is no feature below it. Similarly, if it is the bottom end-vertex of a vertical edge, we define that there is no feature above it.

The function returns a past-the-end iterator for its output sequence.

Requirements

OutputIterator::value_type must be *pair<Arrangement_2::Vertex_const_handle, pair<Object, Object> >*.

CGAL::Arr_observer<Arrangement>

Definition

Arr_observer<Arrangement> serves as an abstract base class for all observer classes that are attached to an arrangement instance of type *Arrangement* and receive notifications from the arrangement. This base class handles the attachment of the observer to a given arrangement instance or to the detachment of the observer from this arrangement instance. It also gives a default empty implementation to all notification functions that are invoked by the arrangement to notify the observer on local or global changes it undergoes. The notification functions are all virtual functions, so they can be overridden by the concrete observer classes that inherit from *Arr_observer<Arrangement>*.

In order to implement a concrete arrangement observer-class, one simply needs to derive from *Arr_observer< Arrangement>* and override the relevant notification functions. For example, if only face-split events are of interest, it is sufficient to override just *before_split_face()* (or just *after_split_face()*).

#include <CGAL/Arr_observer.h>

Types

Arr_observer<Arrangement>:: Arrangement_2 — the type of the associated arrangement.

typedef typename Arrangement_2::Point_2

Point_2; — the point type.

typedef typename Arrangement_2::X_monotone_curve_2

X_monotone_curve_2;

the x-monotone curve type.

typedef typename Arrangement_2::Vertex_handle

Vertex_handle;

typedef typename Arrangement_2::Halfedge_handle

Halfedge_handle;

typedef typename Arrangement_2::Face_handle

Face_handle;

typedef typename Arrangement_2::Ccb_halfedge_circulator

Ccb_halfedge_circulator;

represents the boundary of a connected component (CCB). In particular, holes are represented by a circulator for their outer CCB.

Creation

Arr_observer<Arrangement> obs; — constructs an observer that is unattached to any arrangement instance.

Arr_observer<Arrangement> obs(Arrangement_2& arr);

constructs an observer and attaches it to the given arrangement *arr*.

Modifiers

void — *obs.attach(Arrangement_2& arr)*

attaches the observer to the given arrangement *arr*.

void — *obs.detach()* — detaches the observer from its arrangement.

Notifications on Global Arrangement Operations

virtual void — *obs.before_assign(Arrangement_2 arr)*

issued just before the attached arrangement is assigned with the contents of another arrangement *arr*.

virtual void — *obs.after_assign()* — issued immediately after the attached arrangement has been assigned with the contents of another arrangement.

virtual void — *obs.before_clear()* — issued just before the attached arrangement is cleared.

virtual void — *obs.after_clear(Face_handle uf)*

issued immediately after the attached arrangement has been cleared, so it now consists only of a the unbounded face *uf*.

virtual void — *obs.before_global_change()*

issued just before a global function starts to modify the attached arrangement. It is guaranteed that no queries (especially no point-location queries) are issued until the termination of the global function is indicated by *after_global_change()*.

virtual void — *obs.after_global_change()*

issued immediately after a global function has stopped modifying the attached arrangement.

Notifications on Attachment or Detachment

virtual void *obs.before_attach(Arrangement_2 arr)*

issued just before the observer is attached to the arrangement instance *arr*.

virtual void *obs.after_attach()* issued immediately after the observer has been attached to an arrangement instance.

virtual void *obs.before_detach()* issued just before the observer is detached from its arrangement instance.

virtual void *obs.after_attach()* issued immediately after the observer has been detached from its arrangement instance.

Notifications on Local Changes in the Arrangement

virtual void *obs.before_create_vertex(Point_2 p)*

issued just before a new vertex that corresponds to the point *p* is created.

virtual void *obs.after_create_vertex(Vertex_handle v)*

issued immediately after a new vertex *v* has been created. Note that the vertex still has no incident edges and is not connected to any other vertex.

virtual void *obs.before_create_boundary_vertex(X_monotone_curve_2 cv, Arr_curve_end ind, Arr_parameter_space ps_x, Arr_parameter_space ps_y)*

issued just before a new vertex at infinity is created, *cv* is the curve incident to the surface boundary, *ind* is the relevant curve-end, *ps_x* is the boundary condition of the vertex in *x* and *ps_y* is the boundary condition of the vertex in *y*.

virtual void *obs.after_create_boundary_vertex(Vertex_handle v)*

issued immediately after a new vertex *v* has been created. Note that the vertex still has no incident edges and is not connected to any other vertex.

virtual void *obs.before_create_edge(X_monotone_curve_2 c, Vertex_handle v1, Vertex_handle v2)*

issued just before a new edge that corresponds to the *x*-monotone curve *c* and connects the vertices *v1* and *v2* is created.

virtual void *obs.after_create_edge(Halfedge_handle e)*

issued immediately after a new edge *e* has been created. The halfedge that is sent to this function is always directed from *v1* to *v2* (see above).

virtual void *obs.before_modify_vertex(Vertex_handle v, Point_2 p)*

issued just before a vertex *v* is modified to be associated with the point *p*.

virtual void *obs.after_modify_vertex(Vertex_handle v)*

issued immediately after an existing vertex *v* has been modified.

virtual void *obs.before_modify_edge(Halfedge_handle e, X_monotone_curve_2 c)*

issued just before an edge *e* is modified to be associated with the *x*-monotone curve *c*.

virtual void *obs.after_modify_edge(Halfedge_handle e)*

issued immediately after an existing edge *e* has been modified.

virtual void *obs.before_split_edge(Halfedge_handle e, Vertex_handle v, X_monotone_curve_2 c1, X_monotone_curve_2 c2)*

issued just before an edge *e* is split into two edges that should be associated with the *x*-monotone curves *c1* and *c2*. The vertex *v* corresponds to the split point, and will be used to separate the two resulting edges.

virtual void *obs.after_split_edge(Halfedge_handle e1, Halfedge_handle e2)*

issued immediately after an existing edge has been split into the two given edges *e1* and *e2*.

virtual void *obs.before_split_fictitious_edge(Halfedge_handle e, Vertex_handle v)*

issued just before a fictitious edge *e* is split into two. The vertex at infinity *v* corresponds to the split point, and will be used to separate the two resulting edges.

virtual void *obs.after_split_fictitious_edge(Halfedge_handle e1, Halfedge_handle e2)*

issued immediately after an existing fictitious edge has been split into the two given fictitious edges *e1* and *e2*.

virtual void *obs.before_split_face(Face_handle f, Halfedge_handle e)*

issued just before a face *f* is split into two, as a result of the insertion of the edge *e* into the arrangement.

virtual void *obs.after_split_face(Face_handle f1, Face_handle f2, bool is_hole)*

issued immediately after the existing face *f1* has been split, such that a portion of it now forms a new face *f2*. The flag *is_hole* designates whether *f2* forms a hole inside *f1*.

virtual void *obs.before_split_outer_ccb(Face_handle f,*
Ccb_halfedge_circulator h,
Halfedge_handle e)

issued just before outer ccb *h* inside a face *f* is split into two, as a result of the removal of the edge *e* from the arrangement.

virtual void *obs.after_split_outer_ccb(Face_handle f,*
Ccb_halfedge_circulator h1,
Ccb_halfedge_circulator h2)

issued immediately after outer ccb the face *f* has been split, resulting in the two holes *h1* and *h2*.

virtual void *obs.before_split_inner_ccb(Face_handle f,*
Ccb_halfedge_circulator h,
Halfedge_handle e)

issued just before inner ccb *h* inside a face *f* is split into two, as a result of the removal of the edge *e* from the arrangement.

virtual void *obs.after_split_inner_ccb(Face_handle f,*
Ccb_halfedge_circulator h1,
Ccb_halfedge_circulator h2)

issued immediately after inner ccb the face *f* has been split, resulting in the two holes *h1* and *h2*.

virtual void *obs.before_add_outer_ccb(Face_handle f, Halfedge_handle e)*

issued just before the edge *e* is inserted as a new outer ccb inside the face *f*.

virtual void *obs.after_add_outer_ccb(Ccb_halfedge_circulator h)*

issued immediately after a new outer ccb *h* has been created. The outer ccb always consists of a single pair of twin halfedges.

virtual void *obs.before_add_inner_ccb(Face_handle f, Halfedge_handle e)*

issued just before the edge *e* is inserted as a new inner ccb inside the face *f*.

virtual void *obs.after_add_inner_ccb(Ccb_halfedge_circulator h)*

issued immediately after a new inner ccb *h* has been created. The inner ccb always consists of a single pair of twin halfedges.

virtual void *obs.before_add_isolated_vertex(Face_handle f, Vertex_handle v)*

issued just before the vertex *v* is inserted as an isolated vertex inside the face *f*.

virtual void *obs.after_add_isolated_vertex(Vertex_handle v)*

issued immediately after the vertex *v* has been set as an isolated vertex.

virtual void *obs.before_merge_edge(Halfedge_handle e1, Halfedge_handle e2, X_monotone_curve_2 c)*

issued just before the two edges *e1* and *e2* are merged to form a single edge that will be associated with the *x*-monotone curve *c*.

virtual void *obs.after_merge_edge(Halfedge_handle e)*

issued immediately after two edges have been merged to form the edge *e*.

virtual void *obs.before_merge_fictitious_edge(Halfedge_handle e1, Halfedge_handle e2)*

issued just before the two fictitious edges *e1* and *e2* are merged to form a single fictitious edge.

virtual void *obs.after_merge_fictitious_edge(Halfedge_handle e)*

issued immediately after two fictitious edges have been merged to form the fictitious edge *e*.

virtual void *obs.before_merge_face(Face_handle f1, Face_handle f2, Halfedge_handle e)*

issued just before the two edges *f1* and *f2* are merged to form a single face, following the removal of the edge *e* from the arrangement.

virtual void *obs.after_merge_face(Face_handle f)*

issued immediately after two faces have been merged to form the face *f*.

virtual void *obs.before_merge_outer_ccb(Face_handle f, Ccb_halfedge_circulator h1, Ccb_halfedge_circulator h2, Halfedge_handle e)*

issued just before two outer ccbs *h1* and *h2* inside the face *f* are merged to form a single connected component, following the insertion of the edge *e* into the arrangement.

virtual void *obs.after_merge_outer_ccb(Face_handle f, Ccb_halfedge_circulator h)*

issued immediately after two outer ccbs have been merged to form a single outer ccb *h* inside the face *f*.

virtual void *obs.before_merge_inner_ccb(Face_handle f, Ccb_halfedge_circulator h1, Ccb_halfedge_circulator h2, Halfedge_handle e)*

issued just before two inner ccbs *h1* and *h2* inside the face *f* are merged to form a single connected component, following the insertion of the edge *e* into the arrangement.

virtual void *obs.after_merge_inner_ccb(Face_handle f, Ccb_halfedge_circulator h)*

issued immediately after two inner ccbs have been merged to form a single inner ccb *h* inside the face *f*.

virtual void *obs.before_move_outer_ccb(Face_handle from_f, Face_handle to_f, Ccb_halfedge_circulator h)*

issued just before the outer ccb *h* is moved from one face to another. This can happen if the face *to_f* containing the outer ccb has just been split from *from_f*.

virtual void *obs.after_move_outer_ccb(Ccb_halfedge_circulator h)*

issued immediately after the outer ccb *h* has been moved to a new face.

virtual void *obs.before_move_inner_ccb(Face_handle from_f, Face_handle to_f, Ccb_halfedge_circulator h)*

issued just before the inner ccb *h* is moved from one face to another. This can happen if the face *to_f* containing the inner ccb has just been split from *from_f*.

virtual void *obs.after_move_inner_ccb(Ccb_halfedge_circulator h)*

issued immediately after the inner ccb *h* has been moved to a new face.

virtual void *obs.before_move_isolated_vertex(Face_handle from_f, Face_handle to_f, Vertex_handle v)*

issued just before the isolated vertex *v* is moved from one face to another. This can happen if the face *to_f* containing the isolated vertex has just been split from *from_f*.

virtual void *obs.after_move_isolated_vertex(Vertex_handle v)*

issued immediately after the isolated vertex *v* has been moved to a new face.

virtual void *obs.before_remove_vertex(Vertex_handle v)*

issued just before the vertex *v* is removed from the arrangement.

virtual void *obs.after_remove_vertex()*

issued immediately after a vertex has been removed (and deleted) from the arrangement.

virtual void *obs.before_remove_edge(Halfedge_handle e)*

issued just before the edge e is removed from the arrangement.

virtual void *obs.after_remove_edge()*

issued immediately after an edge has been removed (and deleted) from the arrangement.

virtual void *obs.before_remove_outer_ccb(Face_handle f, Ccb_halfedge_circulator h)*

issued just before the outer ccb f is removed from inside the face f.

virtual void *obs.after_remove_outer_ccb(Face_handle f)*

issued immediately after a outer ccb has been removed (and deleted) from inside the face f.

virtual void *obs.before_remove_inner_ccb(Face_handle f, Ccb_halfedge_circulator h)*

issued just before the inner ccb f is removed from inside the face f.

virtual void *obs.after_remove_inner_ccb(Face_handle f)*

issued immediately after a inner ccb has been removed (and deleted) from inside the face f.

Concept

ArrangementOpenBoundaryTraits_2

Definition

Several predicates are required to handle x-monotone curves that approach infinity and thus approach the boundary of the parameter space. These predicates are sufficient to handle not only curves embedded in an unbounded parameter space, but also curves embedded in a bounded parameter space with open boundaries. Models of the concept ArrangementOpenBoundaryTraits_2 handle curves that approach the boundary of a parameter space. This concept refines the concept *ArrangementBasicTraits_2*. The arrangement template instantiated with a traits class that models this concept can handle x-monotone curves that are unbounded in any direction. The concept ArrangementOpenBoundaryTraits_2, nontheless, also supports planar x-monotone curves that reach the boundary of an open yet bounded parameter space.

An x-monotone curve may be *closed*, in which case its endpoints are representable as *Point_2* objects, or *open* at the boundary of the parameter space. It can have one open end and one closed end (e.g., a ray). The nature of the x-monotone curves, whether they are expected to be closed or not at any one of the four boundary-sides, is conveyed through the definition of the four nested types *Left_side_category*, *Right_side_category*, *Bottom_side_category*, and *Top_side_category*. If some curves handled by a model of the concept ArrangementOpenBoundaryTraits_2 are expected to be open on the left, the nested type *Left_side_category* must be convertible to *Arr_open_side_tag*. Similarly, if some curves handled by the concept are expected to be open on the right, open at the bottom, or open at the top, the corresponding nested type must be convertible to *Arr_open_side_tag*. A model of the concept ArrangementOpenBoundaryTraits_2 must have all the four categories convertible to *Arr_open_side_tag*.[22] In this case the DCEL of the arrangement instantiated with the model is initialized with an implicit bounding rectangle. When the parameter space is bounded, it is the exact geometric embedding of the implicit bounding rectangle.

Refines

ArrangementBasicTraits_2

Categories

ArrangementOpenBoundaryTraits_2:: Left_side_category

Must be convertible to either *Arr_oblivious_side_tag* or *Arr_open_side_tag*.

ArrangementOpenBoundaryTraits_2:: Bottom_side_category

Must be convertible to either *Arr_oblivious_side_tag* or *Arr_open_side_tag*.

ArrangementOpenBoundaryTraits_2:: Top_side_category

Must be convertible to either *Arr_oblivious_side_tag* or *Arr_open_side_tag*.

[22]We intend to introduce more concepts that require only a subset of the categories to be convertible to *Arr_open_side_tag*.

ArrangementOpenBoundaryTraits_2:: Right_side_category

Must be convertible to either *Arr_oblivious_side_tag* or *Arr_open_side_tag*.

Functor Types

ArrangementOpenBoundaryTraits_2:: Parameter_space_in_x_2

models the concept *ArrTraits::ParameterSpaceInX_2*. Required only if the traits class supports unbounded curves that approach the left or the right sides (the *Left_side_category* or the *Right_side_category* categories are convertible to *Arr_open_side_tag*).

ArrangementOpenBoundaryTraits_2:: Compare_y_near_boundary_2

models the concept *ArrTraits::CompareYNearBoundary_2*. Required only if the traits class supports unbounded curves that approach the left or the right sides (the *Left_side_category* or the *Right_side_category* categories are convertible to *Arr_open_side_tag*).

ArrangementOpenBoundaryTraits_2:: Parameter_space_in_y_2

models the concept *ArrTraits::ParameterSpaceInY_2*. Required only if the traits class supports unbounded curves that approach the bottom or the top sides (the *Bottom_side_category* or the *Top_side_category* categories are convertible to *Arr_open_side_tag*).

ArrangementOpenBoundaryTraits_2:: Compare_x_at_limit_2

models the concept *ArrTraits::CompareXAtLimit_2*. Required only if the traits class supports unbounded curves that approach the bottom or the top sides (the *Bottom_side_category* or the *Top_side_category* categories are convertible to *Arr_open_side_tag*).

ArrangementOpenBoundaryTraits_2:: Compare_x_near_limit_2

models the concept *ArrTraits::CompareXNearLimit_2*. Required only if the traits class supports unbounded curves that approach the bottom or the top sides (the *Bottom_side_category* or the *Top_side_category* categories are convertible to *Arr_open_side_tag*).

Accessing Functor Objects

Parameter_space_in_x_2 *traits.parameter_space_in_x_2_object() const*

Compare_y_near_boundary_2

traits.compare_y_near_boundary_2_object() const

Parameter_space_in_y_2 *traits.parameter_space_in_y_2_object() const*

Compare_x_at_limit_2	*traits.compare_x_at_limit_2_object() const*
Compare_x_near_limit_2	*traits.compare_x_near_limit_2_object() const*

Has Models

CGAL::Arr_linear_traits_2<Kernel>
CGAL::Arr_rational_function_traits_2<AlgebraicKernel_d_1>
CGAL::Arr_algebraic_segment_traits_2<Coefficient>
CGAL::Arr_curve_data_traits_2<Tr,XData,Mrg,CData,Cnv>
CGAL::Arr_consolidated_curve_data_traits_2<Traits,Data>

See Also

ArrangementBasicTraits_2 (page 2255)
ArrangementXMonotoneTraits_2 (page 2273)
ArrangementLandmarkTraits_2 (page 2269)
ArrangementTraits_2 (page 2279)

ArrTraits::ParameterSpaceInX_2

Concept

Refines

AdaptableBinaryFunction

Has Models

ArrangementOpenBoundaryTraits_2::Parameter_space_in_x_2

Arr_parameter_space

fo(ArrTraits::X_monotone_curve_2 xcv, Arr_curve_end ce)

Given an x-monotone curve *xcv* and an enumeration *ce* that specifies either the minimum or the maximum end of the curve, determines the location of the curve end along the x-dimension. The variable *xcv* identifies the parametric curve $C(t) = (X(t), Y(t))$ defined over an open or half-open interval with endpoints 0 and 1. The enumeration *ce* identifies an open end $d \in \{0,1\}$ of C. Formally, determines whether $\lim_{t \to d} X(t)$ evaluates to b_l, b_r, or a value in between, where b_l and b_r are the x-coordinates of the left and right boundaries of the parameter space, respectively. Returns *ARR_LEFT_BOUNDARY*, *ARR_RIGHT_BOUNDARY*, or *ARR_INTERIOR*, accordingly.
Precondition: If *ArrTraits::Left_side_category* is not convertible to *Arr_open_side_tag* then $ce \neq$ *ARR_MIN_END*.
Precondition: If *ArrTraits::Right_side_category* is not convertible to *Arr_open_side_tag* then $ce \neq$ *ARR_MAX_END*.
Postcondition: If *ce* = *ARR_MIN_END* then the result is either *ARR_LEFT_BOUNDARY* or *ARR_INTERIOR*.
Postcondition: If *ce* = *ARR_MAX_END* then the result is either *ARR_RIGHT_BOUNDARY* or *ARR_INTERIOR*.

Concept

ArrTraits::CompareYNearBoundary_2

Refines

AdaptableTernaryFunction

Has Models

ArrangementOpenBoundaryTraits_2::Compare_y_near_boundary_2

Comparison_result *fo.operator()(ArrTraits::X_monotone_curve_2 xcv1, ArrTraits::X_monotone_curve_2 xcv2, Arr_curve_end ce)*

Given two x-monotone curves *xcv1* and *xcv2* and an enumeration *ce* that specifies either the minimum or the maximum ends of the curves, compares the y-coordinate of the curves near their respective ends. Returns *SMALLER*, *EQUAL*, or *LARGER* accordingly. More precisely, compares the y-coordinates of the vertical projection of a point p onto *xcv1* and *xcv2*. If *ce* is *ARR_MIN_END*, the predicate *Parameter_space_in_x_2* evaluates to *ARR_LEFT_BOUNDARY* when applied to *xcv1* and *ce* and when applied to *xcv2* and *ce*. In this case p is located far to the left, such that the result is invariant under a translation of p farther to the left. If *ce* is *ARR_MAX_END*, the predicate *Parameter_space_in_x_2* evaluates to *ARR_RIGHT_BOUNDARY* when applied to *xcv1* and *ce* and when applied to *xcv2* and *ce*. In that case p is located far to the right in a similar manner.
Precondition: If *ArrTraits::Left_side_category* is not convertible to *Arr_open_side_tag* then *ce* $\neq$ *ARR_MIN_END*.
Precondition: If *ArrTraits::Right_side_category* is not convertible to *Arr_open_side_tag* then *ce* $\neq$ *ARR_MAX_END*.
Precondition: *parameter_space_in_x_2*(*xcv2*, *ce*) = *parameter_space_in_x_2*(*xcv1*, *ce*).
Precondition: *parameter_space_in_x_2*(*xcv1*, *ce*) $\neq$ *ARR_INTERIOR*.
Precondition: If *parameter_space_in_x_2*(*xcv1*, *ce*) = *ARR_LEFT_BOUNDARY* then *ce* = *ARR_MIN_END*.
Precondition: If *parameter_space_in_x_2*(*xcv1*, *ce*) = *ARR_RIGHT_BOUNDARY* then *ce* = *ARR_MAX_END*.

ArrTraits::ParameterSpaceInY_2

Concept

Refines

AdaptableBinaryFunction

Has Models

ArrangementOpenBoundaryTraits_2::Parameter_space_in_y_2

Arr_parameter_space

fo(ArrTraits::X_monotone_curve_2 xcv, Arr_curve_end ce)

Given an x-monotone curve *xcv* and an enumeration *ce* that specifies either the minimum or the maximum end of the curve, determines the location of the curve end along the y-dimension. The variable *xcv* identifies the parametric curve $C(t) = (X(t), Y(t))$ defined over an open or half-open interval with endpoints 0 and 1. The enumeration *ce* identifies an open end $d \in \{0, 1\}$ of C. Formally, determines whether $\lim_{t \to d} Y(t)$ evaluates to b_b, b_t, or a value in between, where b_b and b_t are the y-coordinates of the bottom and top boundaries of the parameter space, respectively. Returns *ARR_BOTTOM_BOUNDARY*, *ARR_TOP_BOUNDARY*, or *ARR_INTERIOR*, accordingly.
Postcondition: If *ArrTraits::Bottom_side_category* is not convertible to *Arr_open_side_tag* then the result is not *ARR_BOTTOM_BOUNDARY*.
Postcondition: If *ArrTraits::Top_side_category* is not convertible to *Arr_open_side_tag* then the result is not *ARR_TOP_BOUNDARY*.

Concept

ArrTraits::CompareXAtLimit_2

Refines

AdaptableFunctor

Has Models

ArrangementOpenBoundaryTraits_2::Compare_x_at_limit_2

Comparison_result *fo(ArrTraits::Point_2 p, ArrTraits::X_monotone_curve_2 xcv, Arr_curve_end ce)*

Given a point p, an x-monotone curve *xcv*, and an enumeration *ce* that specifies either the minimum or the maximum end of the curve where the curve has a vertical asymptote, compares the x-coordinate of p and the x-coordinate of the limit of the curve at its specificed end. The variable *xcv* identifies the parametric curve $C(t) = (X(t), Y(t))$ defined over an open or half-open interval with endpoints 0 and 1. The enumeration *ce* identifies an open end $d \in \{0,1\}$ of C. Formally, compares the x-coordinate of p and $\lim_{t \to d} X(t)$. Returns *SMALLER*, *EQUAL*, or *LARGER* accordingly.
Precondition: *parameter_space_in_y_2*(*xcv*, *ce*) $\neq$ *ARR_INTERIOR*.
Precondition: If the parameter space is unbounded, C has a vertical asymptote at its d-end; that is, *parameter_space_in_x_2*(*xcv*, *ce*) = *ARR_INTERIOR*.

Comparison_result *fo.operator()(ArrTraits::X_monotone_curve_2 xcv1,*
Arr_curve_end ce1,
ArrTraits::X_monotone_curve_2 xcv2,

Arr_curve_end ce2)

Given two x-monotone curves *xcv1* and *xcv2* and two indices *ce1* and *ce2* that specify either the minimum or the maximum ends of *xcv1* and *xcv2*, respectively, where the curves have vertical asymptotes, compares the x-coordinates of the limits of the curves at their specificed ends. The variables *xcv1* and *xcv2* identify the parametric curves $C_1(t) = (X_1(t), Y_1(t))$ and $C_2(t) = (X_2(t), Y_2(t))$, respectively, defined over open or half-open intervals with endpoints 0 and 1. The indices *ce1* and *ce2* identify open ends $d_1 \in \{0, 1\}$ and $d_2 \in \{0, 1\}$ of C_1 and C_2, respectively. Formally, compares $\lim_{t \to d_1} X_1(t)$ and $\lim_{t \to d_2} X_2(t)$. Returns *SMALLER*, *EQUAL*, or *LARGER* accordingly.
Precondition: *parameter_space_in_y_2*(*xcv1*, *ce1*) $\neq$ *ARR_INTERIOR*.
Precondition: *parameter_space_in_y_2*(*xcv2*, *ce2*) $\neq$ *ARR_INTERIOR*.
Precondition: If the parameter space is unbounded, C_1 has a vertical asymptote at its respective end; that is,
parameter_space_in_x_2(*xcv1*, *ce1*) = *ARR_INTERIOR*.
Precondition: If the parameter space is unbounded, C_2 has a vertical asymptote at its respective end; that is,
parameter_space_in_x_2(*xcv2*, *ce2*) = *ARR_INTERIOR*.

Concept

ArrTraits::CompareXNearLimit_2

Refines

AdaptableTernaryFunction

Has Models

ArrangementOpenBoundaryTraits_2::Compare_x_near_limit_2

Comparison_result *fo.operator()(ArrTraits::X_monotone_curve_2 xcv1, ArrTraits::X_monotone_curve_2 xcv2, Arr_curve_end ce)*

Given two x-monotone curves *xcv1* and *xcv2* and an enumeration *ce* that specifies either the minimum ends or the maximum ends of the curves where the curves have a vertical asymptote, compares the x-coordinate of the curves near their respective ends. Returns *SMALLER*, *EQUAL*, or *LARGER* accordingly. More precisely, compares the x-coordinates of the horizontal projection of a point p onto *xcv1* and *xcv2*. If *xcv1* and *xcv2* approach the bottom boundary-side, p is located far to the bottom, such that the result is invariant under a translation of p farther to the bottom. If *xcv1* and *xcv2* approach the top boundary-side, p is located far to the top in a similar manner.
Precondition: The x-coordinates of the limits of the curves at their respective ends are equal. That is, *compare_x_at_limit_2*(*xcv1*, *xcv2*, *ce*) = *EQUAL*.
Precondition: *parameter_space_in_y_2*(*xcv1*, *ce*) = *parameter_space_in_y_2*(*xcv2*, *ce*).
Precondition: *parameter_space_in_y_2*(*xcv1*, *ce*) $\neq$ *ARR_INTERIOR*.

CGAL::Arrangement_with_history_2<Traits,Dcel>

Definition

An object *arr* of the class *Arrangement_with_history_2<Traits,Dcel>* represents the planar subdivision induced by a set of input curves $\mathcal{C}$. The arrangement is represented as a doubly-connected edge-list (DCEL). As is the case for the *Arrangement_2<Traits,Dcel>*, each DCEL vertex is associated with a point and each edge is associated with an x-monotone curve whose interior is disjoint from all other edges and vertices. Each such x-monotone curve is a subcurve of some $C \in \mathcal{C}$ — or may represent an overlap among several curves in $\mathcal{C}$.

The *Arrangement_with_history_2<Traits,Dcel>* class-template extends the *Arrangement_2* class-template by keeping an additional container of input curves representing $\mathcal{C}$, and by maintaining a cross-mapping between these curves and the arrangement edges they induce. This way it is possible to determine the inducing curve(s) of each arrangement edge. This mapping also allows the traversal of input curves, and the traversal of edges induced by each curve.

The *Arrangement_with_history_2<Traits,Dcel>* template has two parameters:

- The *Traits* template-parameter should be instantiated with a model of the *ArrangementTraits_2* concept. The traits class defines the *Curve_2* type, which represents an input curve. It also defines the types of x-monotone curves and two-dimensional points, namely *X_monotone_curve_2* and *Point_2*, respectively, and supports basic geometric predicates on them.
- The *Dcel* template-parameter should be instantiated with a class that is a model of the *ArranagementDcelWithRebind* concept. The value of this parameter is by default *Arr_default_dcel<Traits>*.

Inherits From

Arrangement_2<Traits,Dcel>

#include <CGAL/Arrangement_with_history_2.h>

Types

typedef Arrangement_with_history_2<Traits_2,Dcel>

Self;	a private type used as an abbreviation of the *Arrangement_with_history_2<Traits,Dcel>* type hereafter.

Arrangement_with_history_2<Traits,Dcel>:: Traits_2

the traits class in use.

Arrangement_with_history_2<Traits,Dcel>:: Dcel

the DCEL representation of the arrangement.

typedef typename Traits_2::Point_2

Point_2;	the point type, as defined by the traits class.

typedef typename Traits_2::X_monotone_curve_2

X_monotone_curve_2;

the x-monotone curve type, as defined by the traits class.

typedef typename Traits_2::Curve_2

Curve_2; the curve type, as defined by the traits class.

In addition, the nested types *Vertex*, *Halfedge* and *Face* are defined, as well as all handle, iterator and circulator types, as defined by the *Arrangement_2* class-template (page 2187).

Arrangement_with_history_2<Traits,Dcel>:: Curve_handle

a handle for an input curve.

Arrangement_with_history_2<Traits,Dcel>:: Curve_iterator

a bidirectional iterator over the curves that induce the arrangement. Its value-type is *Curve_2*.

Arrangement_with_history_2<Traits,Dcel>:: Induced_edge_iterator

an iterator over the edges induced by an input curve. Its value type is *Halfedge_handle*.

Arrangement_with_history_2<Traits,Dcel>:: Originating_curve_iterator

an iterator for the curves that originate a given arrangement edge. Its value type is *Curve_handle*.

Creation

Arrangement_with_history_2<Traits,Dcel> arr;

constructs an empty arrangement containing one unbounded face, which corresponds to the whole plane.

Arrangement_with_history_2<Traits,Dcel> arr(Self other);

copy constructor.

*Arrangement_with_history_2<Traits,Dcel> arr(Traits_2 *traits);*

constructs an empty arrangement that uses the given *traits* instance for performing the geometric predicates.

Assignment Methods

Self&	*arr = other*	assignment operator.
void	*arr.assign(Self other)*	
		assigns the contents of another arrangement.
void	*arr.clear()*	clears the arrangement.

Access Functions

See the *Arrangement_2* reference pages (page 2187) for the full list.

• *Accessing the Input Curves:*

Size	*arr.number_of_curves() const*	
		returns the number of input curves that induce the arrangement.
Curve_iterator	*arr.curves_begin()*	returns the begin-iterator of the curves inducing the arrangement.
Curve_iterator	*arr.curves_end()*	returns the past-the-end iterator of the curves inducing the arrangement.
Size	*arr.number_of_induced_edges(Curve_handle ch) const*	
		returns the number of arrangement edges induced by the curve *ch*.
Induced_edge_iterator		
	arr.induced_edges_begin(Curve_handle ch) const	
		returns the begin-iterator of the edges induced by the curve *ch*.
Induced_edge_iterator		
	arr.induced_edges_end(Curve_handle ch) const	
		returns the past-the-end iterator of the edges induced by the curve *ch*.
Size	*arr.number_of_originating_curves(Halfedge_handle e) const*	
		returns the number of input curves that originate the edge *e*.

Originating_curve_iterator

arr.originating_curves_begin(Halfedge_handle e) const

returns the begin-iterator of the curves originating the edge *e*.

Originating_curve_iterator

arr.originating_curves_end(Halfedge_handle e) const

returns the past-the-end iterator of the curves originating the edge *e*.

Modifiers

See the *Arrangement_2* reference pages (page 2187) for the full list of functions for modifying arrangement vertices.

• *Modifying Arrangement Edges:*

The following functions override their counterparts in the *Arrangement_2* class, as they also maintain the cross-relationships between the input curves and the edges they induce.

Halfedge_handle *arr.split_edge(Halfedge_handle e, Point_2 p)*

splits the edge *e* into two edges (more precisely, into two halfedge pairs), at a given split point *p*. The function returns a handle for the halfedge whose source is the same as *e->source()* and whose target vertex is the split point.
Precondition: *p* lies in the interior of the curve associated with *e*.

Halfedge_handle *arr.merge_edge(Halfedge_handle e1, Halfedge_handle e2)*

merges the edges represented by *e1* and *e2* into a single edge. The function returns a handle for one of the merged halfedges.
Precondition: *e1* and *e2* share a common end-vertex, of degree 2, and the *x*-monotone curves associated with *e1* and *e2* are mergeable into a single *x*-monotone curves.

Face_handle *arr.remove_edge(Halfedge_handle e,*
bool remove_source = true,
bool remove_target = true)

removes the edge *e* from the arrangement. Since the *e* may be the only edge incident to its source vertex (or its target vertex), this vertex can be removed as well. The flags *remove_source* and *remove_target* indicate whether the endpoints of *e* should be removed, or whether they should be left as isolated vertices in the arrangement. If the operation causes two faces to merge, the merged face is returned. Otherwise, the face to which the edge was incident is returned.

See Also

ArrangementDcel (page 2232)
Arr_default_dcel<Traits> (page 2247)
ArrangementTraits_2 (page 2279)
Arrangement_2<Traits,Dcel> (page 2187)
insertion functions (page 2212)
removal functions (page 2375)
overlaying arrangements (page 2221)

Enum

CGAL::Arr_parameter_space

#include <CGAL/Arr_enums.h>

enum Arr_parameter_space { *ARR_LEFT_BOUNDARY,*
ARR_RIGHT_BOUNDARY,
ARR_BOTTOM_BOUNDARY,
ARR_TOP_BOUNDARY,
ARR_INTERIOR}

CGAL::remove_curve

Function

Definition

The function *remove_curve* removes a given curve from a given arrangement.

#include <CGAL/Arrangement_with_history_2.h>

template <class Traits, class Dcel>
Size remove_curve(Arrangement_with_history_2<Traits,Dcel>& arr,
typename Arrangement_with_history_2<Traits,Dcel>::Curve_handle ch)

remove_curve removes a curve, specified by its handle *ch*, from the arrangement *arr*, by deleting all the edges it induces. The function returns the number of deleted edges.

Concept

ArrangementWithHistoryInputFormatter

A model for the ArrangementWithHistoryInputFormatter concept supports a set of functions that enable reading an arrangement-with-history instance from an input stream using a specific format.

Refines

ArrangementInputFormatter

Types

ArrangementWithHistoryInputFormatter:: Arr_with_history_2

the type of arrangement to input.

typedef typename Arrangement_2::Curve_2

Curve_2; the inducing curve type.

Formatted Input Functions

void *inf.read_curves_begin()*

reads a message indicating the beginning of the inducing curves.

void *inf.read_curves_end()*

reads a message indicating the end of the inducing curves.

void *inf.read_curve_begin()*

reads a message indicating the beginning of a single curve record.

void *inf.read_curve_end()* reads a message indicating the end of a single curve record.

void *inf.read_curve(Curve_2& c)*

reads a curve.

void *inf.read_induced_edges_begin()*

reads a message indicating the beginning of the set of edges induced by the current curve.

void *inf.read_induced_edges_end()*

reads a message indicating the end of the induced edges set.

Has Models

Arr_with_history_text_formatter<ArrFormatter> (page 2380)

ArrangementWithHistoryOutputFormatter

A model for the ArrangementWithHistoryOutputFormatter concept supports a set of functions that enable writing an arrangement-with-history instance to an output stream using a specific format.

Refines

ArrangementOutputFormatter

Types

ArrangementWithHistoryOutputFormatter:: Arr_with_history_2

the type of arrangement to output.

typedef typename Arrangement_2::Curve_2

Curve_2; — the inducing curve type.

Formatted Output Functions

void *outf.write_curves_begin()*

writes a message indicating the beginning of the inducing curves.

void *outf.write_curves_end()*

writes a message indicating the end of the inducing curves.

void *outf.write_curve_begin()*

writes a message indicating the beginning of a single curve record.

void *outf.write_curve_end()*

writes a message indicating the end of a single curve record.

void *outf.write_curve(Curve_2 c)*

writes a curve.

void *outf.write_induced_edges_begin()*

writes a message indicating the beginning of the set of edges induced by the current curve.

void *outf.write_induced_edges_end()*

writes a message indicating the end of the induced edges set.

Has Models

Arr_with_history_text_formatter<ArrFormatter> (page 2380)

Class

CGAL::Arr_with_history_text_formatter<ArrFormatter>

Definition

Arr_with_history_text_formatter<ArrFormatter> defines the format of an arrangement in an input or output stream (typically a file stream), thus enabling reading and writing an arrangement-with-history instance using a simple text format.

The *ArrFormatter* parameter servers as a base class for *Arr_with_history_text_formatter<ArrFormatter>* and must be a model of the *ArrangementInputFormatter* and the *ArrangementOutputFormatter* concepts. It is used to read or write the base arrangement, while the derived class is responsible for reading and writing the set of curves inducing the arrangement and maintaining the relations between these curves and the edges they induce.

#include <CGAL/IO/Arr_with_history_text_formatter.h>

Is Model for the Concepts

ArrangementWithHistoryInputFormatter
ArrangementWithHistoryOutputFormatter

See Also

read (page 2228)
write (page 2230)

CGAL::Arr_oblivious_side_tag

Class

Definition

The categories *Left_side_category*, *Right_side_category*, *Bottom_side_category*, and *Top_side_category*, nested in any model of the *ArrangementBasicTraits_2*, must be convertible to *Arr_oblivious_side_tag*. *Arr_oblivious_side_tag* is an empty construct used for dispatching functions based on type of curves that induce the arrangement.

#include <CGAL/Arr_tags.h>

See Also

Arr_open_side_tag(page 2382)
ArrangementBasicTraits_2(page 2255)

Class

CGAL::Arr_open_side_tag

Definition

All the four types *Left_side_category*, *Right_side_category*, *Bottom_side_category*, and *Top_side_category* nested in any model of the concept *ArrangementOpenBoundaryTraits* must be convertible to *Arr_open_side_tag*, which derives from *Arr_oblivious_side_tag*. It implies that some curves are expected to approach the left, right, bottom, or top sides of the open boundary of the parameter space. *Arr_open_side_tag* is an empty construct used for dispatching functions based on type of curves that induce the arrangement.

#include <CGAL/Arr_tags.h>

See Also

Arr_oblivious_side_tag(page 2381)
ArrangementOpenBoundaryTraits_2(page 2360)

Chapter 33

2D Intersection of Curves

Baruch Zukerman, Ron Wein, and Efi Fogel

Contents

33.1 Introduction

Let $\mathcal{C} = \{C_1, C_2, \ldots, C_n\}$ be a set of curves. We wish to compute all intersection points between two curves in the set in an output-sensitive manner, without having to go over all $O(n^2)$ curve pairs. To this end, we sweep an imaginary line l from $x = -\infty$ to $x = \infty$ over the plane. While sweeping the plane, we keep track of the order of curves intersecting it. This order changes at a finite number of *event points*, such that we only have to calculate the intersection points between two curves when they become contiguous. For more details on the *sweep-line algorithm* see, for example, [dBvKOS00, Chapter 2].

This chapter describes three functions implemented using the sweep-line algorithm: given a collection of input curves, compute all intersection points, compute the set of subcurves that are pairwise interior-disjoint induced by them, and checking whether there is at least one pair of curves among them that intersect in their interior.

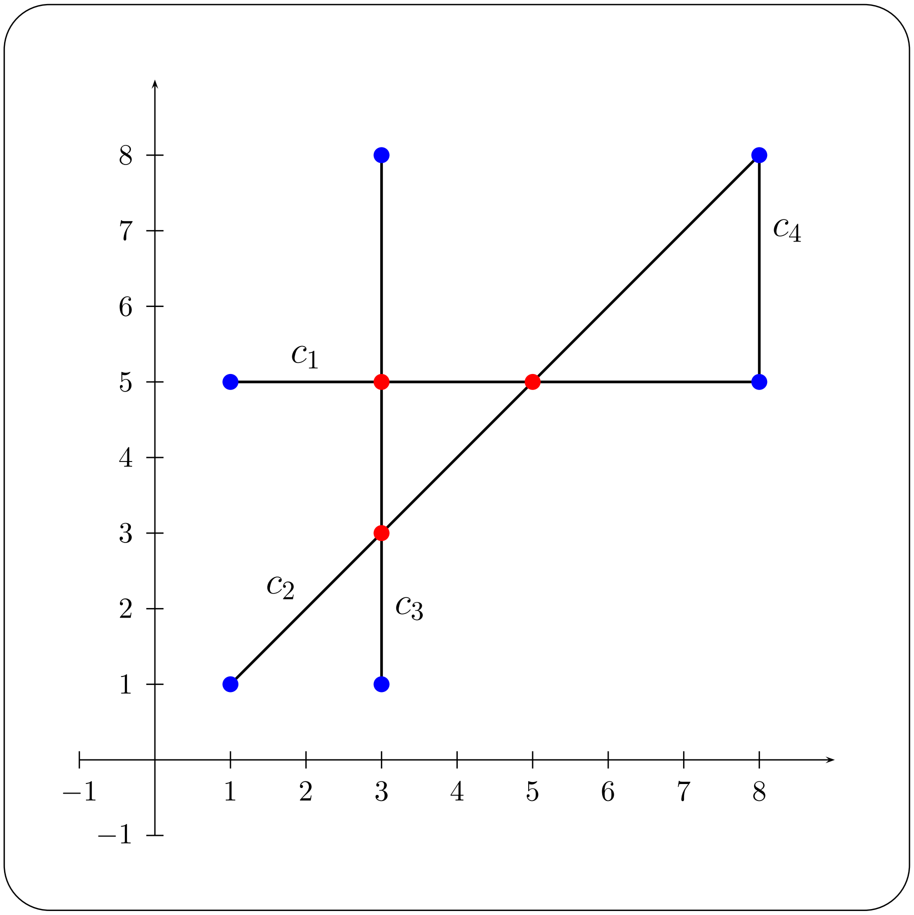
The implementation is robust. It supports general curves and handles all degenerate cases, including overlapping curves, vertical segments, and tangency between curves. The robustness of the algorithm is guaranteed if the functions are instantiated with a traits class that employs certified computations. This traits class must be a model of the *ArrangementTraits_2* concept — see the Chapter 32 for more details.

The complexity of the sweep-line algorithm is $O((n+k)\log n)$ where n is the number of the input curves and k is the number of intersection points induced by these curves.

33.2 Example

The simple program listed below computes intersection points induced by a set of four input segments illustrated in figure 33.1.

```
#include <CGAL/Cartesian.h>
#include <CGAL/MP_Float.h>
#include <CGAL/Quotient.h>
#include <CGAL/Arr_segment_traits_2.h>
#include <CGAL/Sweep_line_2_algorithms.h>
#include <list>

typedef CGAL::Quotient<CGAL::MP_Float>                       NT;
typedef CGAL::Cartesian<NT>                                  Kernel;
typedef Kernel::Point_2                                      Point_2;
typedef CGAL::Arr_segment_traits_2<Kernel>                   Traits_2;
typedef Traits_2::Curve_2                                    Segment_2;

int main()
{
  // Construct the input segments.
  Segment_2 segments[] = {Segment_2 (Point_2 (1, 5), Point_2 (8, 5)),
                          Segment_2 (Point_2 (1, 1), Point_2 (8, 8)),
                          Segment_2 (Point_2 (3, 1), Point_2 (3, 8)),
                          Segment_2 (Point_2 (8, 5), Point_2 (8, 8))};

  // Compute all intersection points.
  std::list<Point_2>     pts;

  CGAL::compute_intersection_points (segments, segments + 4,
                                     std::back_inserter (pts));

  // Print the result.
  std::cout << "Found " << pts.size() << " intersection points: " <<
   std::endl;
  std::copy (pts.begin(), pts.end(),
             std::ostream_iterator<Point_2>(std::cout, "\n"));

  // Compute the non-intersecting sub-segments induced by the input
   segments.
  std::list<Segment_2>   sub_segs;

  CGAL::compute_subcurves(segments, segments + 4,
   std::back_inserter(sub_segs));

  std::cout << "Found " << sub_segs.size()
            << " interior-disjoint sub-segments." << std::endl;

  CGAL_assertion (CGAL::do_curves_intersect (segments, segments + 4));

  return 0;
}
```



8
7
6
5
4
3
2
1
−1
−1
1
2
3
4
5
6
7
8
c_1
c_2
c_3
c_4

File: `examples/Arrangement_on_surface_2/sweep_line.cpp`

33.3 Design and Implementation History

The current version of the sweep-line algorithm was written by Baruch Zukerman, based on previous implementations by Ester Ezra and Tali Zvi.

2D Intersection of Curves Reference Manual

Baruch Zukerman and Ron Wein

This chapter describes three functions implemented using the sweep-line algorithm: given a collection $\mathcal{C}$ of planar curves, compute all intersection points among them, obtain the set of maximal pairwise interior-disjoint subcurves of the curves in $\mathcal{C}$, or check whether there is at least one pair of curves in $\mathcal{C}$ that intersect in their interior.

The first two operations are performed in an output-sensitive manner.

Functions

33.4 Alphabetical List of Reference Pages

Function

CGAL::compute_intersection_points

#include <CGAL/Sweep_line_2_algorithms.h>

template <class InputIterator, class OutputIterator, class Traits>
OutputIterator compute_intersection_points(InputIterator curves_begin,
InputIterator curves_end,
OutputIterator points,
bool report_endpoints = false,
Traits traits = Default_traits())

given a range of curves, compute all intersection points between two (or more) input curves. When the flag *report_endpoints* is *true*, this function reports all the curve endpoints as well. If a curve endpoint is also an intersection point, it is reported once (regardless of the value of the *report_endpoints* flag). The *Traits* type must be a model of the *ArrangementTraits_2* concept, such that the value-type of *InputIterator* is *Traits::Curve_2*, and the value-type of *OutputIterator* is *Traits::Point_2*. The output points are reported in an increasing *xy*-lexicographical order.

CGAL::compute_subcurves

#include <CGAL/Sweep_line_2_algorithms.h>

template <class InputIterator, class OutputIterator, class Traits>
OutputIterator *compute_subcurves(InputIterator curves_begin,*
InputIterator curves_end,
OutputIterator subcurves,
bool multiple_overlaps = false,
Traits traits = Default_traits())

given a range of curves, compute all x-monotone subcurves that are pairwise disjoint in their interior, as induced by the input curves. If the flag *multiple_overlaps* is *true*, then a subcurve that represents an overlap of k input curves is reported k times; otherwise, each subcurve is reported only once. The *Traits* type must be a model of the *ArrangementTraits_2* concept, such that the value-type of *InputIterator* is *Traits::Curve_2*, and the value-type of *OutputIterator* is *Traits::X_monotone_curve_2*.

Function

Function

CGAL::do_curves_intersect

#include <CGAL/Sweep_line_2_algorithms.h>

template <class InputIterator, class Traits>
bool *do_curves_intersect(InputIterator curves_begin, InputIterator curves_end, Traits traits = Default_traits())*

given a range of curves, check whether there is at least one pair of curves that intersect in their interior. The function returns *true* if such a pair is found, and *false* if all curves are pairwise disjoint in their interior.The *Traits* type must be a model of the *ArrangementTraits_2* concept, such that the value-type of *InputIterator* is *Traits::Curve_2*.

Chapter 34

2D Snap Rounding

Eli Packer

Contents

34.1 Introduction

Snap Rounding (SR, for short) is a well known method for converting arbitrary-precision arrangements of segments into a fixed-precision representation [GGHT97, GM98, Hob99]. In the study of robust geometric computing, it can be classified as a finite precision approximation technique. Iterated Snap Rounding (ISR, for short) is a modification of SR in which each vertex is at least half-the-width-of-a-pixel away from any non-incident edge [HP02]. This package supports both methods. Algorithmic details and experimental results are given in [HP02].

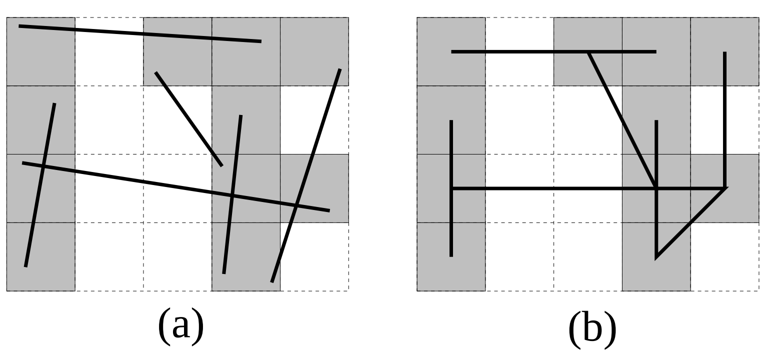


Figure 34.1: An arrangement of segments before (a) and after (b) SR (hot pixels are shaded)

34.2 What is Snap Rounding/Iterated Snap Rounding

Given a finite collection $\mathcal{S}$ of segments in the plane, the arrangement of $\mathcal{S}$ denoted $\mathcal{A}(\mathcal{S})$ is the subdivision of the plane into vertices, edges, and faces induced by $\mathcal{S}$. A *vertex* of the arrangement is either a segment endpoint or the intersection of two segments. Given an arrangement of segments whose vertices are represented with arbitrary-precision coordinates, the SR procedure *snap_rounding_2<Traits,InputIterator,OutputContainer>* proceeds as follows. We tile the plane with a grid of unit squares, *pixels*, each centered at a point with integer coordinates. A pixel is *hot* if it contains a vertex of the arrangement. Each vertex of the arrangement is replaced by the center of the hot pixel containing it and each edge e is replaced by the polygonal chain through the centers of the hot pixels met by e, in the same order as they are met by e. Figure 34.1 demonstrates the results of SR.

In a snap-rounded arrangement, the distance between a vertex and a non-incident edge can be extremely small compared with the width of a pixel in the grid used for rounding. ISR is a modification of SR which makes a vertex and a non-incident edge well separated (the distance between each is at least half-the-width-of-a-pixel). However, the guaranteed quality of the approximation in ISR degrades. Figure 34.2 depicts the results of SR and ISR on the same input. Conceptually, the ISR procedure is equivalent to repeated application of SR, namely we apply SR to the original set of segments, then we use the output of SR as input to another round of SR and so on until all the vertices are well separated from non-incident edges. Algorithmically we operate differently, as this repeated application of SR would have resulted in an efficient overall process. The algorithmic details are given in [HP02].

34.3 Terms and Software Design

Our package supports both schemes, implementing the algorithm described in [HP02]. Although the paper only describes an algorithm for ISR, it is easy to derive an algorithm for SR, by performing only the first rounding level for each segment.

The input to the program is a set S of n segments, $S = \{s_1, \ldots, s_n\}$ and the output is a set G of n polylines, with a polyline g_i for each input segments s_i. An input segment is given by the coordinates of its endpoints. An output polyline is given by the ordered set of vertices $v_0, \ldots, v_k$ along the polyline. The polyline consists of the segments $(v_0 v_1), \ldots, (v_{k-1} v_k)$.

There are three template parameters: *Traits* is the underlying geometry, i.e., the number type used and the coordinate representation. *InputIterator* is the type of the iterators that point to the first and after-the-last elements of the input. Finally, *OutputContainer* is the type of the output container.

Since the algorithm requires kernel functionalities such as the rounding to the center of a pixel, a special traits class must be provided. The precise description of the requirements is given by the concept *SnapRoundingTraits_2*. The class *Snap_rounding_traits_2* is a model of this concept.

34.4 Four Line Segment Example

The following example generates an ISR representation of an arrangement of four line segments. In particular it produces a list of points that are the vertices of the resulting polylines in a plane tiled with one-unit square pixels.

```
#include <CGAL/Cartesian.h>
#include <CGAL/Quotient.h>
```

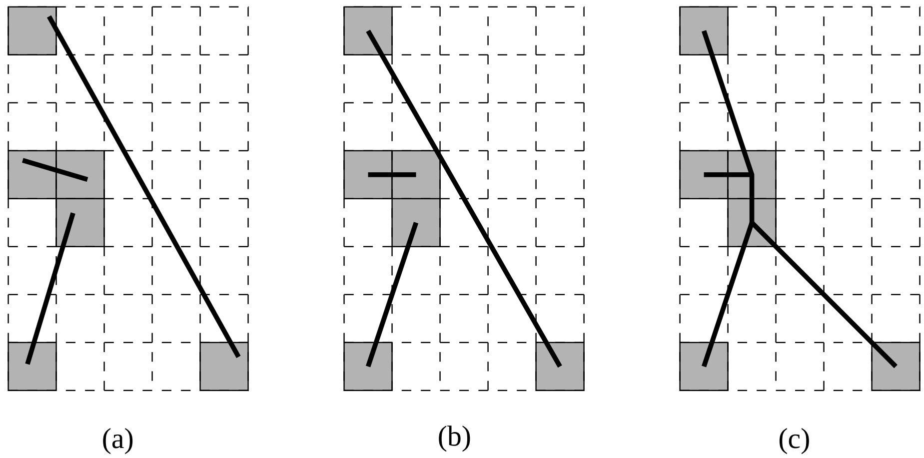


Figure 34.2: An arrangement of segments before (a), after SR (b) and ISR (c) (hot pixels are shaded).

```
#include <CGAL/MP_Float.h>
#include <CGAL/Snap_rounding_traits_2.h>
#include <CGAL/Snap_rounding_2.h>

typedef CGAL::Quotient<CGAL::MP_Float>          Number_type;
typedef CGAL::Cartesian<Number_type>            Kernel;
typedef CGAL::Snap_rounding_traits_2<Kernel>    Traits;
typedef Kernel::Segment_2                       Segment_2;
typedef Kernel::Point_2                         Point_2;
typedef std::list<Segment_2>                    Segment_list_2;
typedef std::list<Point_2>                      Polyline_2;
typedef std::list<Polyline_2>                   Polyline_list_2;

int main()
{
  Segment_list_2 seg_list;
  Polyline_list_2 output_list;

  seg_list.push_back(Segment_2(Point_2(0, 0), Point_2(10, 10)));
  seg_list.push_back(Segment_2(Point_2(0, 10), Point_2(10, 0)));
  seg_list.push_back(Segment_2(Point_2(3, 0), Point_2(3, 10)));
  seg_list.push_back(Segment_2(Point_2(7, 0), Point_2(7, 10)));

  // Generate an iterated snap-rounding representation, where the centers
    of
  // the hot pixels bear their original coordinates, using 5 kd trees:

   CGAL::snap_rounding_2<Traits,Segment_list_2::const_iterator,Polyline_list_2>
    (seg_list.begin(), seg_list.end(), output_list, 1.0, true, false, 5);

  int counter = 0;
  Polyline_list_2::const_iterator iter1;
  for (iter1 = output_list.begin(); iter1 != output_list.end(); ++iter1)
   {
    std::cout << "Polyline number " << ++counter << ":\n";
    Polyline_2::const_iterator iter2;
```

```
    for (iter2 = iter1->begin(); iter2 != iter1->end(); ++iter2)
      std::cout << "    (" << iter2->x() << ":" << iter2->y() << ")\n";
  }

  return(0);
}
```

File: `examples/Snap_rounding_2/snap_rounding.cpp`

This program generates four polylines, one for each input segment. The exact output follows:

```
Polyline number 1:
    (0/4:0/4)
    (12/4:12/4)
    (20/4:20/4)
    (28/4:28/4)
    (40/4:40/4)
Polyline number 2:
    (0/4:40/4)
    (12/4:28/4)
    (20/4:20/4)
    (28/4:12/4)
    (40/4:0/4)
Polyline number 3:
    (12/4:0/4)
    (12/4:12/4)
    (12/4:28/4)
    (12/4:40/4)
Polyline number 4:
    (28/4:0/4)
    (28/4:12/4)
    (28/4:28/4)
    (28/4:40/4)
```

The package is supplied with a graphical demo program that opens a window, allows the user to edit segments dynamically, applies a selected snap-rounding procedures, and displays the result onto the same window (see *<CGAL_ROOT>/demo/Snap_rounding_2/demo.cpp*).

2D Snap Rounding Reference Manual

Eli Packer

Snap Rounding is a method for converting arbitrary-precision arrangements of segments into a fixed-precision representation.

34.5 Alphabetical List of Reference Pages

Function

CGAL::snap_rounding_2

Definition

Snap Rounding (SR, for short) is a well known method for converting arbitrary-precision arrangements of segments into a fixed-precision representation [GGHT97, GM98, Hob99]. In the study of robust geometric computing, it can be classified as a finite precision approximation technique. Iterated Snap Rounding (ISR, for short) is a modification of SR in which each vertex is at least half-the-width-of-a-pixel away from any non-incident edge [HP02]. This package supports both methods. Algorithmic details and experimental results are given in [HP02].

Given a finite collection $\mathcal{S}$ of segments in the plane, the arrangement of $\mathcal{S}$ denoted $\mathcal{A}(\mathcal{S})$ is the subdivision of the plane into vertices, edges, and faces induced by $\mathcal{S}$. A *vertex* of the arrangement is either a segment endpoint or the intersection of two segments. Given an arrangement of segments whose vertices are represented with arbitrary-precision coordinates, SR proceeds as follows. We tile the plane with a grid of unit squares, *pixels*, each centered at a point with integer coordinates. A pixel is *hot* if it contains a vertex of the arrangement. Each vertex of the arrangement is replaced by the center of the hot pixel containing it and each edge e is replaced by the polygonal chain through the centers of the hot pixels met by e, in the same order as they are met by e.

In a snap-rounded arrangement, the distance between a vertex and a non-incident edge can be extremely small compared with the width of a pixel in the grid used for rounding. ISR is a modification of SR which makes a vertex and a non-incident edge well separated (the distance between each is at least half-the-width-of-a-pixel). However, the guaranteed quality of the approximation in ISR degrades. For more details on ISR see [HP02].

The traits used here must support (arbitrary-precision) rational number type as this is a basic requirement of SR.

#include <CGAL/Snap_rounding_2.h>

```
template < class Traits, class InputIterator, class OutputContainer >
void                  snap_rounding_2( InputIterator begin,
                                       InputIterator end,
                                       OutputContainer& output_container,
                                       typename Traits::FT pixel_size,
                                       bool do_isr = true,
                                       bool int_output = true,
                                       unsigned int number_of_kd_trees = 1)
```

The first two parameters denote the first and after-the-last iterators of the input segments. The third parameter is a reference to a container of the output polylines. Since a polyline is composed of a sequence of points, a polyline is a container itself. The fifth parameter determines whether to apply ISR or SR.

The fourth parameter denotes the pixel size w. The plane will be tiled with square pixels of width w such that the origin is the center of a pixel. The sixth parameter denotes the output representation. If the value of the sixth parameter is *true* then the centers of pixels constitute the integer grid, and hence the vertices of the output polylines will be integers. For example, the coordinates of the center of the pixel to the right of the pixel containing the origin will be $(1,0)$ regardless of the pixel width. If the value of the sixth parameter is *false*, then the centers of hot pixels (and hence the vertices of the output polylines) will bear their original coordinates, which may not necessarily be integers. In the latter case, the coordinates of the center of the pixel to the right of the pixel containing the origin, for example, will be $(w,0)$.

The seventh (and last) parameter is briefly described next; for a detailed description see [HP02].

advanced

A basic query used in the algorithm is to report the hot pixels of size w that a certain segment s intersects. An alternative way to do the same is to query the hot pixels' centers contained in a Minkowski sum of s with a pixel of width w centered at the origin; we denote this Minkowski sum by $M(s)$. Since efficiently implementing this kind of query is difficult, we use an orthogonal range-search structure instead. We query with the bounding box $B(M(s))$ of $M(s)$ in a two-dimensional kd-tree which stores the centers of hot pixels. Since $B(M(s))$ in general is larger than $M(s)$, we still need to filter out the hot pixels which do not intersect s.

While this approach is easy to implement with CGAL, it may incur considerable overhead since the area of $B(M(s))$ may be much larger than the area of $M(s)$, possibly resulting in many redundant hot pixels to filter out. Our heuristic solution, which we describe next, is to use a cluster of kd-trees rather than just one. The cluster includes several kd-trees, each has the plane, and hence the centers of hot pixels, rotated by a different angle in the first quadrant of the plane; for our purpose, a rotation by angles outside this quadrant is symmetric to a rotation by an angle in the first quadrant.

Given a parameter c, the angles of rotation are $(i-1)\frac{\pi}{2c}, i = 1,\ldots,c$, and we construct a kd-tree corresponding to each of these angles. Then for a query segment s, we choose the kd-tree for which the area of $B(M(s))$ is the smallest, in order to (potentially) get less hot pixels to filter out. Since constructing many kd-trees may be costly, our algorithm avoids building a kd-tree which it expects to be queried a relatively small number of times (we estimate this number in advance). How many kd-trees should be used? It is difficult to provide a simple answer for that. There are inputs for which the time to build more than one kd-tree is far greater than the time saved by having to filter out less hot pixels (sparse arrangements demonstrate this behavior), and there are inputs which benefit from using several kd-trees. Thus, the user can control the number of kd-trees with the parameter *number_of_kd_trees*. Typically, but not always, one kd-tree (the default) is sufficient.

advanced

Precondition: *pixel_size* must have a positive value and *number_of_kd_trees* must be a positive integer.

SnapRoundingTraits_2

Definition

The concept SnapRoundingTraits_2 lists the set of requirements that must be fulfilled by an instance of the *Traits* template-parameter of the function *snap_rounding_2<Traits,InputIterator,OutputContainer>()*. This concept provides the types of the geometric primitives used in this class and some function object types for the required predicates on those primitives.

Refines

This concept refines the standard concepts DefaultConstructible, Assignable and CopyConstructible. It also refines the concept *SweepLineTraits_2* (page **??**). An instance of this concept is used as the traits class for the *Sweep_line_2.get_intersection_points()* operation. The requirements listed below are induced by compnents of the *snap_rounding_2()* function other than the call to *Sweep_line_2.get_intersection_points()*. Naturally, some of them may already be listed in *SweepLineTraits_2*.

Types

SnapRoundingTraits_2:: FT	The number type. This type must fulfill the requirements on *FieldNumberType*
SnapRoundingTraits_2:: Point_2	The point type.
SnapRoundingTraits_2:: Segment_2	The segment type.
SnapRoundingTraits_2:: Iso_rectangle_2	The iso-rectangle type.
SnapRoundingTraits_2:: Construct_vertex_2	Function object. Must provide the operator *Point_2 operator()(Segment_2 seg, int i)*, which returns the source or target of *seg*. If *i* modulo 2 is 0, the source is returned, otherwise the target is returned.
SnapRoundingTraits_2:: Construct_segment_2	Function object. Must provide the operator *Segment_2 operator()(Point_2 p, Point_2 q)*, which introduces a segment with source *p* and target *q*. The segment is directed from the source towards the target.
SnapRoundingTraits_2:: Construct_iso_rectangle_2	Function object. Must provide the operator *Iso_rectangle_2 operator()(Point_2 left, Point_2 right, Point_2 bottom, Point_2 top)*, which introduces an iso-oriented rectangle fo whose minimal *x* coordinate is the one of *left*, the maximal *x* coordinate is the one of *right*, the minimal *y* coordinate is the one of *bottom*, the maximal *y* coordinate is the one of *top*.
SnapRoundingTraits_2:: To_double	Function object. Must provide the operator *double operator()(FT)*, which computes an approximation of a given number of type *FT*. The precision of this operation is of not high significance, as it is only used in the implementation of the heuristic technique to exploit a cluster of kd-trees rather than just one.

SnapRoundingTraits_2:: Compare_x_2	Function object. Must provide the operator *Comparison_result operator()(Point_2 p, Point_2 q)* which returns *SMALLER, EQUAL* or *LARGER* according to the x-ordering of points p and q.
SnapRoundingTraits_2:: Compare_y_2	Function object. Must provide the operator *Comparison_result operator()(Point_2 p, Point_2 q)* which returns *SMALLER, EQUAL* or *LARGER* according to the y-ordering of points p and q.
SnapRoundingTraits_2:: Snap_2	Rounds a point to a center of a pixel (unit square) in the grid used by the Snap Rounding algorithm. Note that no conversion to an integer grid is done yet. Must have the syntax *void operator()(Point_2 p,FT pixel_size,FT &x,FT &y)* where p is the input point, *pixel_size* is the size of the pixel of the grid, and x and y are the x and y-coordinates of the rounded point respectively.
SnapRoundingTraits_2:: Integer_grid_point_2	Convert coordinates into an integer representation where one unit is equal to pixel size. For instance, if a point has the coordinates $(3.7,5.3)$ and the pixel size is 0.5, then the new point will have the coordinates of $(7,10)$. Note, however, that the number type remains the same here, although integers are represented. Must have the syntax *Point_2 operator()(Point_2 p,NT pixel_size)* where p is the converted point and *pixel_size* is the size of the pixel of the grid.

SnapRoundingTraits_2:: Minkowski_sum_with_pixel_2

Returns the vertices of a polygon, which is the Minkowski sum of a segment and a square centered at the origin with edge size *pixel edge*. Must have the syntax *void operator()(std::list<Point_2>& vertices_list, Segment_2 s, NT unit_square)* where *vertices_list* is the list of the vertices of the Minkowski sum polygon, s is the input segment and *unit_square* is the edge size of the pixel.

Creation

This concept refines the standard concepts DefaultConstructible, Assignable and CopyConstructible.

Operations

The following functions construct the required function objects occasionally referred as functors listed above.

Construct_vertex_2	*traits.construct_vertex_2_object()*
Construct_segment_2	*traits.construct_segment_2_object()*
Construct_iso_rectangle_2	*traits.construct_iso_rectangle_2_object()*
Compare_x_2	*traits.compare_x_2_object()*
Compare_y_2	*traits.compare_y_2_object()*
Snap_2	*traits.snap_2_object()*
Integer_grid_point_2	*traits.integer_grid_point_2_object()*
Minkowski_sum_with_pixel_2	*traits.minkowski_sum_with_pixel_2_object()*

Has Models

CGAL::Snap_rounding_traits<Kernel>

See Also

CGAL::Snap_rounding_2<Traits>

CGAL::Snap_rounding_traits_2<Kernel>

The class *Snap_rounding_traits_2<Kernel>* is a model of the *SnapRoundingTraits_2* concept, and is the only traits class supplied with the package. This class should be instantiated with an exact geometric kernel that conforms to the CGAL kernel-concept, such as the *Cartesian<gmpq>* kernel.

This geometric kernel must provide an (arbitrary-precision) rational number type (*FT*), *Point_2*, *Segment_2* and *Iso_rectangle_2*. It should be possible to cast numbers of the number type *FT* to double-precision representation. That is, the function *CGAL::to_double(FT)* must be supported.

The *CGAL::to_double()* function is used to implement the operation that rounds the coordinates of a point to a center of a pixel. This operation is one of the traits-concept requirement. The coordinates are converted to double, rounded down to the nearest grid point, and finally adjusted to lie on a center of a pixel. Notice that if *CGAL::to_double()* returns the closet value, then when it rounds up a given coordinate, the resulting ISR, may be imprecise, and the distance between some vertex and a non-incident edge can be slightly less than the guaranteed half-the width-of-a-pixel.

#include <CGAL/Snap_rounding_traits_2.h>

Is Model for the Concepts

SnapRoundingTraits_2

Chapter 35

Envelopes of Curves in 2D

Ron Wein

Contents

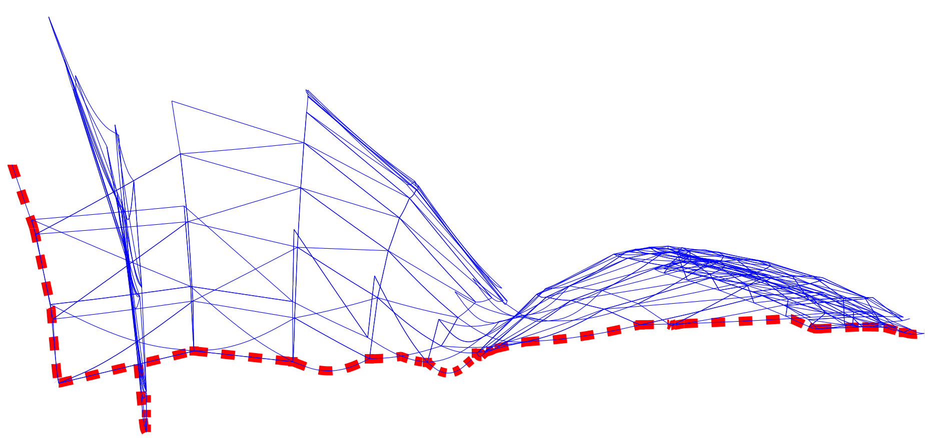


Figure 35.1: The lower envelope of a set of line segments and hyperbolic arc.

35.1 Introduction

A continuous curve C in $\mathbb{R}^2$ is called *x-monotone*, if every vertical line intersects it at a single point at most. For example, the circle $x^2+y^2=1$ is *not* xy-monotone as the vertical line $x=0$ intersects it at $(0,-1)$ and at $(0,1)$;

however, it is possible to split the circle into an upper part and a lower part, such that both of these parts are x-monotone. We consider vertical segments as *weakly* x-monotone, to properly handle inputs that contain such vertical curves.

An x-monotone curve can be represented as a univariate function $y = C(x)$, defined over some continuous range $R_C \subseteq \mathbb{R}$. Given a set $\mathcal{C} = \{C_1, C_2, \ldots, C_n\}$ of x-monotone curves, their *lower envelope* is defined as the point-wise minimum of all curves. Namely, the lower envelope of the set $\mathcal{C}$ can be defined as the following function:

$$\mathcal{L}_{\mathcal{C}}(x) = \min_{1 \leq k \leq n} \overline{C}_k(x) \ ,$$

where we define $\overline{C}_k(x) = C_k(x)$ for $x \in R_{C_k}$, and $\overline{C}_k(x) = \infty$ otherwise.

Similarly, the *upper envelope* of $\mathcal{C}$ is the point-wise maximum of the x-monotone curves in the set:

$$\mathcal{U}_{\mathcal{C}}(x) = \max_{1 \leq k \leq n} \underline{C}_k(x) \ ,$$

where in this case $\underline{C}_k(x) = -\infty$ for $x \notin R_{C_k}$.

Given a set of x-monotone curves $\mathcal{C}$, the *minimization diagram* of $\mathcal{C}$ is a subdivision of the x-axis into cells, such that the identity of the curves that induce the lower envelope over a specific cell of the subdivision (an edge or a vertex) is the same. In non-degenerate situations, an edge — which represents a continuous interval on the x-axis — is induced by a single curve (or by no curves at all, if there are no x-monotone curves defined over the interval), and a vertex is induced by a single curve and corresponds to one of its endpoints, or by two curves and corresponds to their intersection point. The *maximization diagram* is symmetrically defined for upper envelopes. In the rest of this chapter, we refer to both these diagrams as *envelope diagrams*.

Lower and upper envelopes can be efficiently computed using a divide-and-conquer approach. First, note that the envelope diagram for a single x-monotone curve C_k is trivial to compute: we project the boundary of its range of definition R_{C_k} onto the x-axis and label the features it induces accordingly. Given a set $\mathcal{D}$ of (non necessarily x-monotone) curves in $\mathbb{R}^2$, we subdivide each curve into a finite number of weakly x-monotone curves, and obtain the set $\mathcal{C}$. Then, we split the set into two disjoint subsets $\mathcal{C}_1$ and $\mathcal{C}_2$, and we compute their envelope diagrams recursively. Finally, we merge the diagrams in linear time by traversing both diagrams in parallel.

35.2 The Envelope Diagram

The package basically contains two sets of free functions: *lower_envelope_x_monotone_2 (begin, end, diag)* (similarly *upper_envelope_x_monotone_2()*) construct the envelope diagram for a given range of x-monotone curves, while *lower_envelope_2 (begin, end, diag)* (similarly *upper_envelope_2()*) construct the envelope diagram for a range of *arbitrary* (not necessarily x-monotone) curves. In this section we explain more on the structure of the envelope diagram these functions output.

A minimization diagram or a maximization diagram is represented by a model of the concept *EnvelopeDiagram_1*. This concept defines the structure of the subdivision of the x-axis into 0-dimensional cells called *vertices*, and 1-dimensional cells called *edges*. The important property of this subdivision is that the identity of the curves that induce the lower envelope (or the upper envelope) over each cell is fixed.

Figure 35.2 shows the lower envelope of a set of eight line segments, and sketches the structure of their minimization diagram. Each diagram vertex v is associated with a point p_v on the envelope, which corresponds to either a curve endpoint or to an intersection point of two (or more) curves. The vertex is therefore associated with a set of x-monotone curves that induce the envelope over p_v. Each vertex is incident to two edges, one lying to its left and the other to its right.

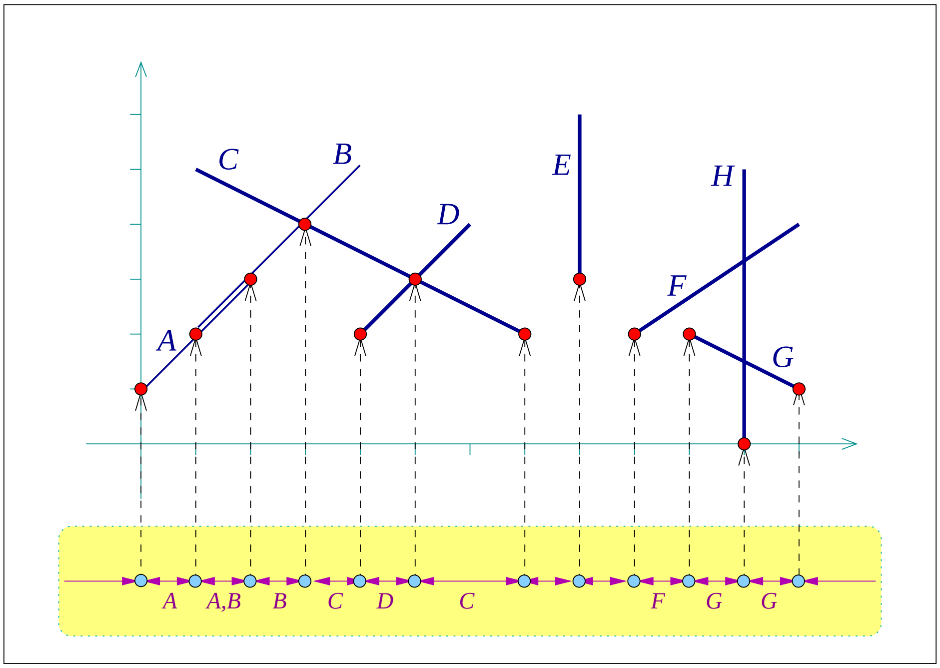


Figure 35.2: The lower envelope of eight line segments, labeled $A, \ldots, H$, as constructed in *envelope_segments.cpp*. The minimization diagram is shown at the bottom, where each diagram vertex points to the point associated with it, and the labels of the segment that induce a diagram edge are displayed below this edge. Note that there exists one edge that represents an overlap (i.e., there are two segments that induce it), and there are also a few edges that represent empty intervals.

An edge in the envelope diagram represents a continuous portion of the x-axis, and is associated with a set of x-monotone curves that induce the envelope over this interval. Note that this set may be empty if no x-monotone curves are defined over this interval. In degenerate situations where curves overlap, there may be more than a single curve that induces the envelope over the interval the edge represents. An envelop diagram of a set of curves either consists of a single unbounded edge (in case the curve set is empty or if the envelope contains a single unbounded curve that is below or above all other curves), or at least one vertex and two unbounded edges, while each additional vertex comes along with an additional edge. It is possible to directly access the *leftmost* edge, representing the unbounded interval that starts at $-\infty$, and the *rightmost* edge, representing the unbounded interval that ends at ∞. (In the example depicted in Figure 35.2 we have only bounded curves, so the leftmost and rightmost edges represent empty intervals. This is not the case when we deal, for example, with envelopes of sets of lines.)

Any model of the concept must define a geometric traits class, which in turn defines the *Point_2* and *X_monotone_curve_2* types defined with the diagram features. The geometric traits class must be a model of the *ArrangementXMonotoneTraits_2* concept in case we construct envelopes of x-monotone curves. If we are interested in handling arbitrary (not necessarily x-monotone) curves, the traits class must be a model of the *ArrangementTraits_2* concept. This concepts refined the *ArrangementXMonotoneTraits_2* concept; a traits class that models this concepts must also defines a *Curve_2* type, representing an arbitrary planar curve, and provide a functor for subdividing such curves into x-monotone subcurves.

35.3 Examples

35.3.1 Example for Envelope of Line Segments

The following example demonstrates how to compute and traverse the minimization diagram of line segments, as illustrated in Figure 35.2. We use the curve-data traits instantiated by the *Arr_segment_traits_2* class, in order to attach a label (a *char* in this case) to each input segment. We use these labels when we print the minimization diagram:

```
#include <CGAL/Gmpq.h>
#include <CGAL/Cartesian.h>
#include <CGAL/Arr_segment_traits_2.h>
#include <CGAL/Arr_curve_data_traits_2.h>
#include <CGAL/Envelope_diagram_1.h>
#include <CGAL/envelope_2.h>

#include <list>
#include <iostream>

typedef CGAL::Gmpq                                        Number_type;
typedef CGAL::Cartesian<Number_type>                      Kernel;
typedef CGAL::Arr_segment_traits_2<Kernel>                Segment_traits_2;
typedef Segment_traits_2::X_monotone_curve_2              Segment_2;
typedef CGAL::Arr_curve_data_traits_2<Segment_traits_2,
                                       char>              Traits_2;
typedef Traits_2::Point_2                                 Point_2;
typedef Traits_2::X_monotone_curve_2
    Labeled_segment_2;
typedef CGAL::Envelope_diagram_1<Traits_2>                Diagram_1;

int main ()
{
  // Consrtuct the input segments and label them 'A' ... 'H'.
  std::list<Labeled_segment_2>   segments;

  segments.push_back (Labeled_segment_2 (Segment_2 (Point_2 (0, 1),
                                                     Point_2 (2, 3)),
    'A'));
  segments.push_back (Labeled_segment_2 (Segment_2 (Point_2 (1, 2),
                                                     Point_2 (4, 5)),
    'B'));
  segments.push_back (Labeled_segment_2 (Segment_2 (Point_2 (1, 5),
                                                     Point_2 (7, 2)),
    'C'));
  segments.push_back (Labeled_segment_2 (Segment_2 (Point_2 (4, 2),
                                                     Point_2 (6, 4)),
    'D'));
  segments.push_back (Labeled_segment_2 (Segment_2 (Point_2 (8, 3),
                                                     Point_2 (8, 6)),
    'E'));
  segments.push_back (Labeled_segment_2 (Segment_2 (Point_2 (9, 2),
                                                     Point_2 (12, 4)),
```

```
    'F'));
  segments.push_back (Labeled_segment_2 (Segment_2 (Point_2 (10, 2),
                                                    Point_2 (12, 1)),
    'G'));
  segments.push_back (Labeled_segment_2 (Segment_2 (Point_2 (11, 0),
                                                    Point_2 (11, 5)),
    'H'));

  // Compute the minimization diagram that represents their lower
    envelope.
  Diagram_1              min_diag;

  lower_envelope_x_monotone_2 (segments.begin(), segments.end(),
                               min_diag);

  // Print the minimization diagram.
  Diagram_1::Edge_const_handle     e = min_diag.leftmost();
  Diagram_1::Vertex_const_handle   v;
  Diagram_1::Curve_const_iterator  cit;

  while (e != min_diag.rightmost())
  {
    std::cout << "Edge:";
    if (! e->is_empty())
    {
      for (cit = e->curves_begin(); cit != e->curves_end(); ++cit)
        std::cout << ' ' << cit->data();
    }
    else
      std::cout << " [empty]";
    std::cout << std::endl;

    v = e->right();
    std::cout << "Vertex (" << v->point() << "):";
    for (cit = v->curves_begin(); cit != v->curves_end(); ++cit)
      std::cout << ' ' << cit->data();
    std::cout << std::endl;

    e = v->right();
  }
  CGAL_assertion (e->is_empty());
  std::cout << "Edge: [empty]" << std::endl;

  return (0);
}
```

File: examples/Envelope_2/envelope_segments.cpp

35.3.2 Example for Computing the Convex Hull with Envelopes

The next example computes the convex hull of a set of input points by constructing envelopes of unbounded curves, in our case lines that are dual to the input points. Here use the *Arr_linear_traits_2* class to compute the lower envelope of the set of dual lines. We read a set of points $\mathcal{P} = p_1, \ldots, p_n$ from an input file, and construct the corresponding dual lines $\mathcal{P}^* = p_1^*, \ldots, p_n^*$, where the line p^* dual to a point $p = (p_x, p_y)$ is given by $y = p_x x - p_y$. We then compute the convex hull of the point-set $\mathcal{P}$, using the fact that the lines that form the lower envelope of $\mathcal{P}^*$ are dual to the points along the *upper* part of $\mathcal{P}$'s convex hull, and the lines that form the upper envelope of $\mathcal{P}^*$ are dual to the points along the *lower* part of the convex hull; see, e.g., [dBvKOS00, Section 11.4] for more details. Note that the leftmost edge of the minimization diagram is associated with the same line as the rightmost edge of the maximization diagram, and vice-verse. We can therefore skip the rightmost edges of both diagrams:

```
#include <CGAL/Gmpq.h>
#include <CGAL/Cartesian.h>
#include <CGAL/Arr_linear_traits_2.h>
#include <CGAL/Arr_curve_data_traits_2.h>
#include <CGAL/Envelope_diagram_1.h>
#include <CGAL/envelope_2.h>
#include <vector>

typedef CGAL::Gmpq                                          Number_type;
typedef CGAL::Cartesian<Number_type>                        Kernel;
typedef CGAL::Arr_linear_traits_2<Kernel>                   Linear_traits_2;
typedef Linear_traits_2::Point_2                            Point_2;
typedef Linear_traits_2::Line_2                             Line_2;
typedef CGAL::Arr_curve_data_traits_2<Linear_traits_2,
                                      unsigned int>         Traits_2;
typedef Traits_2::X_monotone_curve_2                        Dual_line_2;
typedef CGAL::Envelope_diagram_1<Traits_2>                  Diagram_1;

int main (int argc, char* argv[])
{
  // Read the points from the input file.
  const char* filename = (argc > 1) ? argv[1] : "ch_points.dat";
  std::ifstream in_file(filename);
  if (!in_file.is_open()) {
    std::cerr << "Failed to open " << filename << " ..." << std::endl;
    return -1;
  }

  // Read the points from the file, and construct their dual lines.
  std::list<Dual_line_2>  dual_lines;

  unsigned int            n;
  in_file >> n;
  std::vector<Point_2>    points;
  points.resize(n);

  for (unsigned int k = 0; k < n; ++k) {
    int px, py;
    in_file >> px >> py;
    points[k] = Point_2 (px, py);

    // The line dual to the point (p_x, p_y) is y = p_x*x - p_y,
    // or: p_x*x - y - p_y = 0:
    Line_2 line = Line_2 (Number_type(px), Number_type(-1),
   Number_type(-py));
```

```
    // Generate the x-monotone curve based on the line and the point
      index.
    dual_lines.push_back (Dual_line_2 (line, k));
  }
  in_file.close();

  // Compute the lower envelope of dual lines, which corresponds to the
      upper
  // part of the convex hull, and their upper envelope, which corresponds
      to
  // the lower part of the convex hull.
  Diagram_1              min_diag;
  Diagram_1              max_diag;
  lower_envelope_x_monotone_2(dual_lines.begin(), dual_lines.end(),
    min_diag);
  upper_envelope_x_monotone_2(dual_lines.begin(), dual_lines.end(),
    max_diag);

  // Output the points along the boundary convex hull in counterclockwise
  // order. We start by traversing the minimization diagram from left to
  // right, then the maximization diagram from right to left.
  std::cout << "The convex hull of " << points.size() << " input
    points:";
  Diagram_1::Edge_const_handle  e = min_diag.leftmost();
  while (e != min_diag.rightmost()) {
    std::cout << " (" << points[e->curve().data()] << ')';
    e = e->right()->right();
  }

  // Handle the degenerate case of a vertical convex hull edge:
  if (e->curve().data() != max_diag.leftmost()->curve().data())
    std::cout << " (" << points[e->curve().data()] << ')';

  e = max_diag.leftmost();
  while (e != max_diag.rightmost()) {
    std::cout << " (" << points[e->curve().data()] << ')';
    e = e->right()->right();
  }
  std::cout << std::endl;

  return 0;
}
```

File: examples/Envelope_2/convex_hull.cpp

35.3.3 Example for Envelope of Non-Linear Curves

We conclude by an example of envelopes of non-linear curves. We use the *Arr_circle_segment_traits_2* class to construct the lower and the upper envelopes of a set of four circles, as depicted in Figure 35.3. Note that unlike

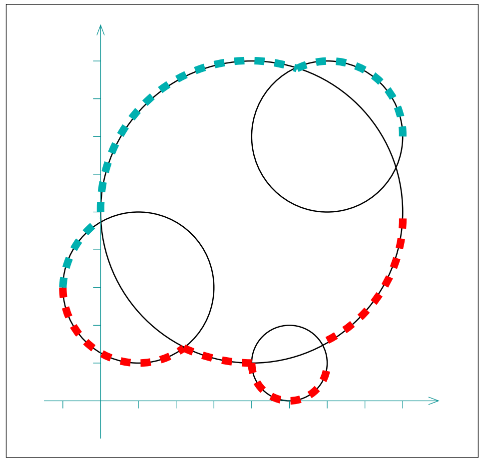


Figure 35.3: A set of four circles, as constructed in *ex_envelope_circles.cpp*. The lower envelope and the upper envelope are shown using thick dashed lines of different colors respectively.

the two previous examples, here our curves are not x-monotone, so we use the functions that compute envelopes of arbitrary curves:

```
#include <CGAL/Gmpq.h>
#include <CGAL/Cartesian.h>
#include <CGAL/Arr_circle_segment_traits_2.h>
#include <CGAL/Arrangement_2.h>
#include <CGAL/Envelope_diagram_1.h>
#include <CGAL/envelope_2.h>

typedef CGAL::Gmpq                                          Number_type;
typedef CGAL::Cartesian<Number_type>                        Kernel;
typedef Kernel::Point_2                                     Kernel_point_2;
typedef Kernel::Circle_2                                    Circle_2;
typedef CGAL::Arr_circle_segment_traits_2<Kernel>           Traits_2;
typedef Traits_2::Curve_2                                   Curve_2;
typedef CGAL::Envelope_diagram_1<Traits_2>                  Diagram_1;

/*! Print the given envelope diagram. */
void print_diagram (const Diagram_1& diag)
{
  Diagram_1::Edge_const_handle     e = diag.leftmost();
  Diagram_1::Vertex_const_handle   v;

  while (e != diag.rightmost())
  {
    std::cout << "Edge: ";
    if (! e->is_empty())
    {
      Circle_2      circ = e->curve().supporting_circle();
      std::cout << " (x - " << CGAL::to_double(circ.center().x()) << ")^2
  +"
                << " (y - " << CGAL::to_double(circ.center().y()) << ")^2
```

```
    = "
                << CGAL::to_double(circ.squared_radius()) << std::endl;
    }
    else
      std::cout << " [empty]" << std::endl;

    v = e->right();
    std::cout << "Vertex (" << CGAL::to_double(v->point().x()) << ' '
              << CGAL::to_double(v->point().y()) << ')' << std::endl;

    e = v->right();
  }
  CGAL_assertion (e->is_empty());
  std::cout << "Edge: [empty]" << std::endl;

  return;
}

/*! The main program. */
int main ()
{
  // Create four input circles.
  Curve_2          circles[4];

  circles[0] = Circle_2 (Kernel_point_2 (1, 3), CGAL::square(2));
  circles[1] = Circle_2 (Kernel_point_2 (4, 5), CGAL::square(4));
  circles[2] = Circle_2 (Kernel_point_2 (5, 1), CGAL::square(1));
  circles[3] = Circle_2 (Kernel_point_2 (6, 7), CGAL::square(2));

  // Compute the minimization diagram that represents their lower
    envelope.
  Diagram_1              min_diag;

  lower_envelope_2 (&(circles[0]), &(circles[4]), min_diag);
  print_diagram (min_diag);

  // Compute the maximization diagram that represents the upper envelope.
  Diagram_1              max_diag;

  upper_envelope_2 (&(circles[0]), &(circles[4]), max_diag);
  print_diagram (max_diag);

  return (0);
}
```

File: examples/Envelope_2/envelope_circles.cpp

Envelopes of Curves in 2D
Reference Manual

Ron Wein

Introduction

This package consists of functions that compute the lower (or upper) envelope of a set of arbitrary curves in 2D. The output is represented as an envelope diagram, namely a subdivision of the x-axis into intervals, such that the identity of the curves that induce the envelope over each interval is unique.

Functions

Concepts

Classes

35.4 Alphabetical List of Reference Pages

CGAL::lower_envelope_2

#include <CGAL/envelope_2.h>

template<class InputIterator, class EnvelopeDiagram>
void *lower_envelope_2(InputIterator begin, InputIterator end, EnvelopeDiagram& diag)*

Computes the lower envelope of a set of curves in $\mathbb{R}^2$, as given by the range *[begin, end)*. The lower envelope is represented using the output minimization diagram *diag*, which must be a model of the *EnvelopeDiagram_1* concept.

Precondition: The value-type of *InputIterator* is *EnvelopeDiagram::Traits_2::Curve_2*.

Function

Function

CGAL::upper_envelope_2

#include <CGAL/envelope_2.h>

template<class InputIterator, class EnvelopeDiagram>
void *upper_envelope_2(InputIterator begin, InputIterator end, EnvelopeDiagram& diag)*

Computes the upper envelope of a set of curves in $\mathbb{R}^2$, as given by the range *[begin, end)*. The upper envelope is represented using the output maximization diagram *diag*, which must be a model of the *EnvelopeDiagram_1* concept.
Precondition: The value-type of *InputIterator* is *EnvelopeDiagram::Traits_2::Curve_2*.

CGAL::lower_envelope_x_monotone_2

#include <CGAL/envelope_2.h>

template<class InputIterator, class EnvelopeDiagram>
void lower_envelope_x_monotone_2(InputIterator begin,
InputIterator end,
EnvelopeDiagram& diag)

Computes the lower envelope of a set of x-monotone curves in $\mathbb{R}^2$, as given by the range *[begin, end)*. The lower envelope is represented using the output minimization diagram *diag*, which must be a model of the *EnvelopeDiagram_1* concept.
Precondition: The value-type of *InputIterator* is *EnvelopeDiagram::X_monotone_curve_2*.

Function

CGAL::upper_envelope_x_monotone_2

#include <CGAL/envelope_2.h>

template<class InputIterator, class EnvelopeDiagram>
void *upper_envelope_x_monotone_2(InputIterator begin, InputIterator end, EnvelopeDiagram& diag)*

Computes the upper envelope of a set of x-monotone curves in $\mathbb{R}^2$, as given by the range *[begin, end)*. The upper envelope is represented using the output maximization diagram *diag*, which must be a model of the *EnvelopeDiagram_1* concept.
Precondition: The value-type of *InputIterator* is *EnvelopeDiagram::X_monotone_curve_2*.

EnvelopeDiagram_1

Concept

Definition

This concept defines the representation of an envelope diagram of a set of planar curve. The *envelope diagram* is a subdivision of the x-axis into 0-dimensional cells (*vertices*) and 1-dimensional cells (*edges*), such that the identity of the curves that induce the lower envelope (or the upper envelope) over each cell is fixed.

A vertex in an envelope diagram is therefore associated with a point on the envelope, and corresponds to either a curve endpoint or to an intersection point of two (or more) curves. Therefore each vertex is associated with a set of x-monotone curves that induce the envelope over this point. Each vertex is incident to two edges, one lying to its left and the other to its right.

An edge in the envelope diagram represents a continuous portion of the x-axis, and is associated with a (possibly empty) set of curves that induce the envelope over this portion of the x-axis. An edge may be bounded by two vertices, one to its left and the other to its right. However, the diagram contains two unbounded edges, its *leftmost* edge, representing the interval $(-\infty, x_l)$, and its *rightmost* edge, representing the interval (x_r, ∞), where x_l and x_r are the x-coodinates of the leftmost and the rightmost vertices in the diagram, respectively. Note that a diagram may contain no vertices at all, in which case it comprises a single edge.

Note that any model of the EnvelopeDiagram_1 concept must define a geometric traits class, which in turn defines the *Point_2* and *X_monotone_curve_2* types defined with the diagram features.

Types

EnvelopeDiagram_1:: Traits_2	the geometric traits class.
typedef Traits_2::Point_2 Point_2;	the point type.
typedef Traits_2::X_monotone_curve_2 X_monotone_curve_2;	the x-monotone curve type.
EnvelopeDiagram_1:: Size	the size type (convertible to *size_t*).
EnvelopeDiagram_1:: Curve_const_iterator	an iterator for the x-monotone curves that induce a diagram feature. Its value-type is *X_monotone_curve_2*.
EnvelopeDiagram_1:: Vertex	the vertex type, a model of the concept *EnvelopeDiagramVertex*.
EnvelopeDiagram_1:: Edge	the edge type, a model of the concept *EnvelopeDiagramEdge*.
EnvelopeDiagram_1:: Vertex_handle	a handle to a diagram vertex.
EnvelopeDiagram_1:: Vertex_const_handle	a non-mutable handle to a diagram vertex.

EnvelopeDiagram_1:: Edge_handle	a handle to a diagram edge.
EnvelopeDiagram_1:: Edge_const_handle	a non-mutable handle to a diagram edge.

Creation

EnvelopeDiagram_1 diag;

constructs an empty diagram containing one unbounded edge, which corresponds to the entire plane and has no x-monotone curves that are associated with it.

EnvelopeDiagram_1 diag(Self other);

copy constructor.

Access Functions

Edge_const_handle *diag.leftmost() const*

returns the leftmost edge of the diagram (a non-const version is also available).

Edge_const_handle *diag.rightmost() const*

returns the rightmost edge of the diagram (a non-const version is also available).

Modifiers

void *diag.set_leftmost(Edge_const_handle e)*

sets the leftmost edge of the diagram to be e.

void *diag.set_rightmost(Edge_const_handle e)*

sets the rightmost edge of the diagram to be e.

Vertex_handle *diag.new_vertex(Point_2 p)*

creates a new diagram vertex, associated with the point p.

Edge_handle *diag.new_edge()*

creates a new diagram edge.

void *diag.delete_vertex(Vertex_handle v)*

deletes the given vertex v.

void *diag.delete_edge(Edge_handle e)*

deletes the given edge e.

See Also

EnvelopeDiagramVertex (page 2421)
EnvelopeDiagramEdge (page 2423)

EnvelopeDiagramVertex

Concept

Definition

A vertex record in an envelope diagram. It is always associated with a point on the lower (upper) envelope of a non-empty set of curves. A vertex is also associated with a set of x-monotone curves that induce the envelope over this point. It is incident to two edges, one lying to its left and the other to its right.

Types

EnvelopeDiagramVertex:: Size	the size type (convertible to *size_t*).
EnvelopeDiagramVertex:: Edge	the corresponding diagram-edge type.
EnvelopeDiagramVertex:: Point_2	the point type associated with the vertex.
EnvelopeDiagramVertex:: X_monotone_curve_2	the x-monotone curve type.

Creation

EnvelopeDiagramVertex v;	default constructor.
EnvelopeDiagramVertex v(other);	copy constructor.
EnvelopeDiagramVertex v(Point_2 p);	constructs a vertex associated with the given point p.

Access Functions

Point_2	*v.point() const*	returns the point associated with v.
Size	*v.number_of_curves() const*	returns the number of x-monotone curves associated with v.
Curve_const_iterator	*v.curves_begin() const*	return an iterator for the first x-monotone curve associated with v.
Curve_const_iterator	*v.curves_end() const*	return a past-the-end iterator for the x-monotone curves associated with v.
Edge_const_handle	*v.left() const*	returns the edge lying to v's left.
Edge_const_handle	*v.right() const*	returns the edge lying to v's right.

Modifiers

void	*v.set_point(Point_2 p)*	
		associates the point p with v.
void	*v.clear_curves()*	clears the set of curves associated with v.
void	*v.add_curve(X_monotone_curve_2 cv)*	
		adds the x-monotone curve cv to the set of curves associated with v.
void	*v.add_curves(Curve_const_iterator begin, Curve_const_iterator end)*	
		adds the given range of x-monotone curves to the set of curves associated with v.
void	*v.set_left(Edge_const_handle e)*	
		sets the edge lying to the left of v to be e.
void	*v.set_right(Edge_const_handle e)*	
		sets the edge lying to the right of v to be e.

See Also

EnvelopeDiagram_1 (page 2419)
EnvelopeDiagramEdge (page 2423)

EnvelopeDiagramEdge

Concept

Definition

An edge record in an envelope diagram, which represents a continuous portion of the x-axis. It is associated with a (possibly empty) set of curves that induce the envelope over this portion of the x-axis. Note that all curves in this set overlap over the interval represented by the edge.

Types

EnvelopeDiagramEdge:: Size — the size type (convertible to *size_t*).

EnvelopeDiagramEdge:: Vertex — the corresponding diagram-vertex type.

EnvelopeDiagramEdge:: X_monotone_curve_2 — the x-monotone curve type.

Creation

EnvelopeDiagramEdge e; — default constructor.

EnvelopeDiagramEdge e(other); — copy constructor.

Access Functions

Size	*e.number_of_curves() const*	returns the number of x-monotone curves associated with e.
bool	*e.is_empty() const*	returns whether e represents an empty interval — namely, whether the set of x-monotone curves associated with it is empty.
X_monotone_curve_2	*e.curve() const*	returns a representative x-monotone curve associated with e. *Precondition*: e does not represent an empty interval.
Curve_const_iterator	*e.curves_begin() const*	return an iterator for the first x-monotone curve associated with e.
Curve_const_iterator	*e.curves_end() const*	return a past-the-end iterator for the x-monotone curves associated with e.
Vertex_const_handle	*e.left() const*	returns the vertex lying to e's left. *Precondition*: e is not the leftmost edge in the diagram.

Vertex_const_handle	*e.right() const*	returns the vertex lying to *e*'s right. *Precondition*: *e* is not the rightmost edge in the diagram.

Modifiers

void	*e.clear_curves()*	clears the set of curves associated with *e*.
void	*e.add_curve(X_monotone_curve_2 cv)*	
		adds the *x*-monotone curve *cv* to the set of curves associated with *e*.
void	*e.add_curves(Curve_const_iterator begin, Curve_const_iterator end)*	
		adds the given range of *x*-monotone curves to the set of curves associated with *e*.
void	*e.set_left(Vertex_const_handle v)*	
		sets the vertex lying to the left of *e* to be *v*.
void	*e.set_right(Vertex_const_handle v)*	
		sets the vertex lying to the right of *e* to be *v*.

See Also

EnvelopeDiagram_1 (page 2419)
EnvelopeDiagramVertex (page 2421)

CGAL::Envelope_diagram_1<Traits>

Class

Definition

This class is the default envelope-diagram class used by envelope functions to represent the minimization or the maximization diagram of a set of curves. It represents the diagram as a doubly-linked list of interleaved vertices and edges. Thus, all operations provided by the envelope diagram take constant time, and the space needed to store the diagram class is linear in the complexity of the envelope.

The envelope-diagram class is parameterized by a traits class, which is a model of the *ArrangementXMonotoneTraits_2* concept, in case we handle only envelopes of x-monotone curves, or of the refined *ArrangementTraits_2* concept in case we handle arbitrary planar curves.

#include <CGAL/Envelope_diagram_1.h>

Is Model for the Concepts

EnvelopeDiagram_1

Chapter 36

Envelopes of Surfaces in 3D

Dan Halperin, Michal Meyerovitch, Ron Wein, and Baruch Zukerman

Contents

36.1 Introduction

A continuous surface S in $\mathbb{R}^3$ is called *xy-monotone*, if every line parallel to the z-axis intersects it at a single point at most. For example, the sphere $x^2+y^2+z^2=1$ is *not* xy-monotone as the z-axis intersects it at $(0,0,-1)$ and at $(0,0,1)$; however, if we use the xy-plane to split it to an upper hemisphere and a lower hemisphere, these two hemispheres are xy-monotone.

An xy-monotone surface can therefore be represented as a bivariate function $z = S(x,y)$, defined over some continuous range $R_S \subseteq \mathbb{R}^2$. Given a set $\mathcal{S} = \{S_1, S_2, \ldots, S_n\}$ of xy-monotone surfaces, their *lower envelope* is defined as the point-wise minimum of all surfaces. Namely, the lower envelope of the set $\mathcal{S}$ can be defined as the following function:

$$\mathcal{L}_{\mathcal{S}}(x,y) = \min_{1 \le k \le n} \overline{S}_k(x,y) ,$$

where we define $\overline{S}_k(x,y) = S_k(x,y)$ for $(x,y) \in R_{S_k}$, and $\overline{S}_k(x,y) = \infty$ otherwise.

Similarly, the *upper envelope* of $\mathcal{S}$ is the point-wise maximum of the xy-monotone surfaces in the set:

$$\mathcal{U}_{\mathcal{S}}(x,y) = \max_{1 \le k \le n} \underline{S}_k(x,y) ,$$

where in this case $\underline{S}_k(x,y) = -\infty$ for $(x,y) \notin R_{S_k}$.

Given a set of xy-monotone surfaces $\mathcal{S}$, the *minimization diagram* of $\mathcal{S}$ is a subdivision of the xy-plane into cells, such that the identity of the surfaces that induce the lower diagram over a specific cell of the subdivision (be it a face, an edge, or a vertex) is the same. In non-degenerate situation, a face is induced by a single surface (or by no surfaces at all, if there are no xy-monotone surfaces defined over it), an edge is induced by a single surface and corresponds to its projected boundary, or by two surfaces and corresponds to their projected intersection curve, and a vertex is induced by a single surface and corresponds to its projected boundary point, or by three surfaces and corresponds to their projected intersection point. The *maximization diagram* is symmetrically defined for upper envelopes. In the rest of this chapter, we refer to both these diagrams as *envelope diagrams*.

It is easy to see that an envelope diagram is no more than a planar arrangement (see Chapter 32), represented using an extended DCEL structure, such that every DCEL record (namely each face, halfedge and vertex) stores an additional container of it originators: the xy-monotone surfaces that induce this feature.

Lower and upper envelopes can be efficiently computed using a divide-and-conquer approach. First note that the envelope diagram for a single xy-monotone curve S_k is trivial to compute: we project the boundary of its range of definition R_{S_k} onto the xy-plane, and label the faces it induces accordingly. Given a set $\mathcal{D}$ of (non necessarily xy-monotone) surfaces in $\mathbb{R}^3$, we subdivide each surface into a finite number of weakly xy-monotone surfaces,[1] and obtain the set $\mathcal{S}$. Then, we split the set into two disjoint subsets $\mathcal{S}_1$ and $\mathcal{S}_2$, and we compute their envelope diagrams recursively. Finally, we merge the diagrams, and we do this by overlaying them and then applying some post-processing on the resulting diagram. The post-processing stage is non-trivial and involves the projection of intersection curves onto the xy-plane — see [Mey06] for more details.

36.2 The Envelope-Traits Concept

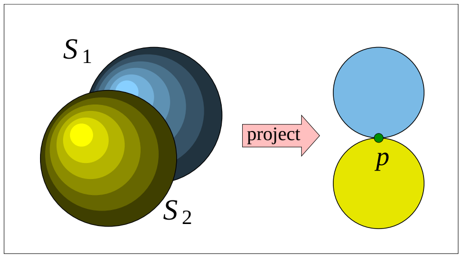
The implementation of the envelope-computation algorithm is generic and can handle arbitrary surfaces. It is parameterized with a traits class, which defines the geometry of surfaces it handles, and supports all the necessary functionality on these surfaces, and on their projections onto the xy-plane. The traits class must model the *EnvelopeTraits_3* concept, the details of which are given below.

As the representation of envelope diagrams is based on 2D arrangements, and the envelop-computation algorithm employs overlay of planar arrangements, the *EnvelopeTraits_3* refines the *ArrangementXMonotoneTraits_2* concept. Namely, a model of this concept must define the planar types *Point_2* and *X_monotone_curve_2* and support basic operations on them, as listed in Section 32.6. Moreover, it must define the spacial types *Surface_3* and *Xy_monotone_surface_3* (in practice, these two types may be the same). Any model of the envelope-traits concept must also support the following operations on these spacial types:

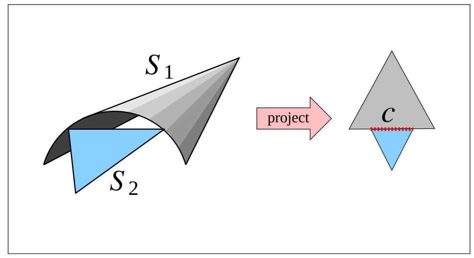
- Subdivide a given surface into continuous xy-monotone surfaces. It is possible to disregard xy-monotone surfaces that do not contribute to the surface envelope at this stage (for example, if we are given a sphere, it is possible to return just its lower hemisphere if we are interested in the lower envelope; the upper hemisphere is obviously redundant).
- Given an xy-monotone surface S, construct all planar curves that form the boundary of the vertical projection S's boundary onto the xy-plane.

  This operation is used at the bottom of the recursion to build the minimization diagram of a single xy-monotone surface.
- Construct all geometric entities that comprise the projection (onto the xy-plane) of the intersection between two xy-monotone surfaces S_1 and S_2. These entities may be:

[1] We consider *vertical* surfaces, namely patches of planes that are perpendicular to the xy-plane, as *weakly* xy-monotone, to handle degenerate inputs properly.



(a)



(b)

Figure 36.1: (a) The spheres S_1 and S_2 have only one two-dimensional point p in their common xy-definition range. They do not necessarily intersect over this point, and the envelope-construction algorithm needs to determine their relative z-order over p. (b) The z-order of the surfaces S_1 and S_2 should be determined over the x-monotone curve c. The comparison is performed over the *interior* of c, excluding its endpoints.

– A planar curve, which is the projection of an 3D intersection curve of S_1 and S_2 (for example, the intersection curve between two spheres is a 3D circle, which becomes an ellipse when projected onto the xy-plane). In many cases it is also possible to indicate the multiplicity of the intersection: if it is odd, the two surfaces intersect transversely and change their relative z-positions on either side of the intersection curve; if it the multiplicity is even, they maintain their relative z-position. Providing the multiplicity information is optional. When provided, it is used by the algorithm to determine the relative order of S_1 and S_2 on one side of their intersection curve when their order on the other side of that curve is known, thus improving the performance of the algorithm.

– A point, induces by the projection of a tangency point of S_1 and S_2, *or* by the projection of a vertical intersection curve onto the xy-plane.

Needless to say, the set of intersection entities may be empty in case S_1 and S_2 do not intersect.

- Given two xy-monotone surfaces S_1 and S_2, and a planar point $p = (x_0, y_0)$ that lies in their common xy-definition range, determine the z-order of S_1 and S_2 over p, namely compare $S_1(x_0, y_0)$ and $S_2(x_0, y_0)$. This operation is used only in degenerate situations, in order to determine the surface inducing the envelope over a vertex (see Figure 36.1(a) for an illustration of a situation when this operation is used).

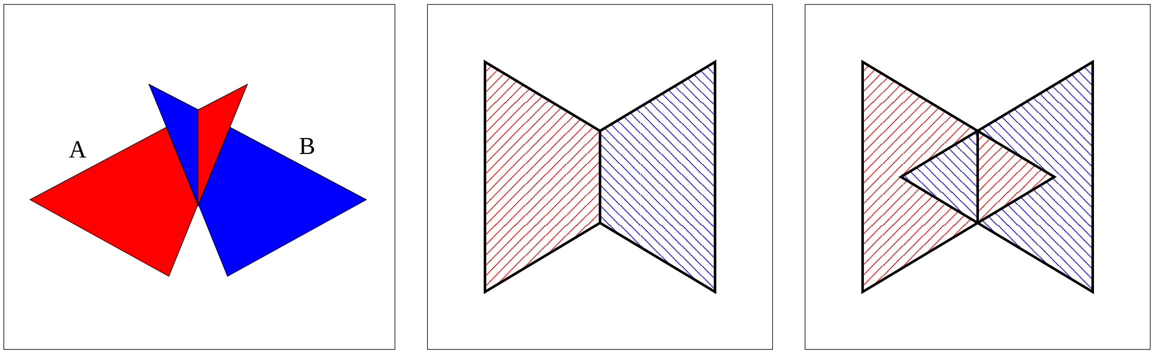
- Given two xy-monotone surfaces S_1 and S_2, and a planar x-monotone curve c, which is a part of their projected intersection, determine the z-order of S_1 and S_2 immediately above (or, similarly, immediately below) the curve c. Note that c is a planar x-monotone curve, and we refer to the region above (or below) it in the *plane*. If c is a vertical curve, we regard the region to its left as lying above it, and the region to its right as lying below it.

  This operation is used by the algorithm to determine the surface that induce the envelope over a face incident to c.

- Given two xy-monotone surfaces S_1 and S_2, and a planar x-monotone curve c, which fully lies in their common xy-definition range, and such that S_1 and S_2 do not intersect over the interior of c, determine the relative z-order of s_1 and s_2 over the interior of c. Namely, we compare $S_1(x_0, y_0)$ and $S_2(x_0, y_0)$ for some point (x_0, y_0) on c.

  This operation is used by the algorithm to determine which surface induce the envelope over an edge associated with the x-monotone curve c, or of a face incident to c, in situations where the previous predicate cannot be used, as c is *not* an intersection curve of S_1 and S_2 (see Figure 36.1(b) for an illustration of a situation where this operation is used).

The package currently contains a traits class for named *Env_triangle_traits_3<Kenrel>* handling 3D triangles, and another named *Env_sphere_traits_3<ConicTraits>* for 3D spheres, based on geometric operations on conic curves (ellipses). In addition, the package includes a traits-class decorator that enables users to attach external (non-geometric) data to surfaces. The usage of the various traits classes is demonstrated in the next section.



(a) (b) (c)

Figure 36.2: (a) Two triangles in $\mathbb{R}^3$, as given in *envelope_triangles.cpp*. (b) Their lower envelope. (c) Their upper envelope.

36.3 Examples

36.3.1 Example for Envelope of Triangles

The following example shows how to use the envelope-traits class for 3D triangles and how to traverse the envelope diagram. It constructs the lower and upper envelopes of the two triangles, as depicted in Figure 36.2(a) and prints the triangles that induce each face and each edge in the output diagrams. For convenience, we use the traits-class decorator *Env_surface_data_traits_3* to label the triangles. When printing the diagrams, we just output the labels of the triangles:

```
#include <CGAL/Gmpq.h>
#include <CGAL/Cartesian.h>
#include <CGAL/Env_triangle_traits_3.h>
#include <CGAL/Env_surface_data_traits_3.h>
#include <CGAL/envelope_3.h>
#include <iostream>
#include <list>

typedef CGAL::Gmpq                                                Number_type;
typedef CGAL::Cartesian<Number_type>                              Kernel;
typedef CGAL::Env_triangle_traits_3<Kernel>                       Traits_3;
typedef Kernel::Point_3                                           Point_3;
typedef Traits_3::Surface_3                                       Triangle_3;
typedef CGAL::Env_surface_data_traits_3<Traits_3, char>           Data_traits_3;
typedef Data_traits_3::Surface_3                                  Data_triangle_3;
typedef CGAL::Envelope_diagram_2<Data_traits_3>
    Envelope_diagram_2;

/* Auxiliary function - print the features of the given envelope diagram.
    */
void print_diagram (const Envelope_diagram_2& diag)
{
  // Go over all arrangement faces.
  Envelope_diagram_2::Face_const_iterator            fit;
  Envelope_diagram_2::Ccb_halfedge_const_circulator  ccb;
```

```
  Envelope_diagram_2::Surface_const_iterator         sit;

  for (fit = diag.faces_begin(); fit != diag.faces_end(); ++fit)
  {
    // Print the face boundary.
    if (fit->is_unbounded())
    {
      std::cout << "[Unbounded face]";
    }
    else
    {
      // Print the vertices along the outer boundary of the face.
      ccb = fit->outer_ccb();
      std::cout << "[Face]  ";
      do
      {
        std::cout << '(' << ccb->target()->point() << ")  ";
        ++ccb;
      } while (ccb != fit->outer_ccb());
    }

    // Print the labels of the triangles that induce the envelope on this
     face.
    std::cout << "-->  " << fit->number_of_surfaces()
              << " triangles:";

    for (sit = fit->surfaces_begin(); sit != fit->surfaces_end(); ++sit)
      std::cout << ' ' << sit->data();
    std::cout << std::endl;
  }

  // Go over all arrangement edges.
  Envelope_diagram_2::Edge_const_iterator            eit;

  for (eit = diag.edges_begin(); eit != diag.edges_end(); ++eit)
  {
    // Print the labels of the triangles that induce the envelope on this
     edge.
    std::cout << "[Edge]  (" << eit->source()->point()
              << ")  (" << eit->target()->point()
              << ")  -->  " << eit->number_of_surfaces()
              << " triangles:";

    for (sit = eit->surfaces_begin(); sit != eit->surfaces_end(); ++sit)
      std::cout << ' ' << sit->data();
    std::cout << std::endl;
  }

  return;
}

/* The main program: */
int main ()
{
```

```
  // Construct the input triangles, makred A and B.
  std::list<Data_triangle_3>   triangles;

  triangles.push_back (Data_triangle_3 (Triangle_3 (Point_3 (0, 0, 0),
                                                     Point_3 (0, 6, 0),
                                                     Point_3 (5, 3, 4)),
                                         'A'));
  triangles.push_back (Data_triangle_3 (Triangle_3 (Point_3 (6, 0, 0),
                                                     Point_3 (6, 6, 0),
                                                     Point_3 (1, 3, 4)),
                                         'B'));

  // Compute and print the minimization diagram.
  Envelope_diagram_2      min_diag;

  CGAL::lower_envelope_3 (triangles.begin(), triangles.end(),
                          min_diag);

  std::cout << std::endl << "The minimization diagram:" << std::endl;
  print_diagram (min_diag);

  // Compute and print the maximization diagram.
  Envelope_diagram_2      max_diag;

  CGAL::upper_envelope_3 (triangles.begin(), triangles.end(),
                          max_diag);

  std::cout << std::endl << "The maximization diagram:" << std::endl;
  print_diagram (max_diag);

  return (0);
}
```

File: examples/Envelope_3/envelope_triangles.cpp

36.3.2 Example for Envelope of Spheres

The next example demonstrates how to instantiate and use the envelope-traits class for spheres, based on the *Arr_conic_traits_2* class that handles the projected intersecion curves. The program reads a set of spheres from an input file and constructs their lower envelope:

```
#include <CGAL/basic.h>

#ifndef CGAL_USE_CORE
#include <iostream>
int main()
{
  std::cout << "Sorry, this example needs CORE ..." << std::endl;
  return 0;
}
#else
```

```
#include <CGAL/Cartesian.h>
#include <CGAL/CORE_algebraic_number_traits.h>
#include <CGAL/Arr_conic_traits_2.h>
#include <CGAL/Env_sphere_traits_3.h>
#include <CGAL/envelope_3.h>
#include <CGAL/Timer.h>
#include <iostream>
#include <list>

typedef CGAL::CORE_algebraic_number_traits            Nt_traits;
typedef Nt_traits::Rational                           Rational;
typedef Nt_traits::Algebraic                          Algebraic;
typedef CGAL::Cartesian<Rational>                     Rat_kernel;
typedef Rat_kernel::Point_3                           Rat_point_3;
typedef CGAL::Cartesian<Algebraic>                    Alg_kernel;

typedef CGAL::Arr_conic_traits_2<Rat_kernel, Alg_kernel, Nt_traits>
                                                      Conic_traits_2;

typedef CGAL::Env_sphere_traits_3<Conic_traits_2>     Traits_3;
typedef Traits_3::Surface_3                           Sphere_3;
typedef CGAL::Envelope_diagram_2<Traits_3>            Envelope_diagram_2;

int main(int argc, char **argv)
{
  // Get the name of the input file from the command line, or use the
   default
  // fan_grids.dat file if no command-line parameters are given.
  const char * filename = (argc > 1) ? argv[1] : "spheres.dat";

  // Open the input file.
  std::ifstream     in_file(filename);

  if (! in_file.is_open())
  {
    std::cerr << "Failed to open " << filename << " ..." << std::endl;
    return 1;
  }

  // Read the spheres from the file.
  // The input file format should be (all coordinate values are
   integers):
  // <n>                          // number of spheres.
  // <x_1> <y_1> <x_1> <R_1>      // center and squared radious of
   sphere #1.
  // <x_2> <y_2> <x_2> <R_2>      // center and squared radious of
   sphere #2.
  //   :     :     :     :
  // <x_n> <y_n> <x_n> <R_n>      // center and squared radious of
   sphere #n.
  int                     n = 0;
  std::list<Sphere_3>     spheres;
  int                     x = 0, y = 0, z = 0, sqr_r = 0;
```

```
  int                     i;

  in_file >> n;
  for (i = 0; i < n; ++i)
  {
    in_file >> x >> y >> z >> sqr_r;
    spheres.push_back(Sphere_3(Rat_point_3(x, y, z), Rational(sqr_r)));
  }
  in_file.close();

  // Compute the lower envelope.
  Envelope_diagram_2    min_diag;
  CGAL::Timer           timer;

  std::cout << "Constructing the lower envelope of "
            << n << " spheres." << std::endl;

  timer.start();
  CGAL::lower_envelope_3(spheres.begin(), spheres.end(), min_diag);
  timer.stop();

  // Print the dimensions of the minimization diagram.
  std::cout << "V = " << min_diag.number_of_vertices()
            << ",  E = " << min_diag.number_of_edges()
            << ",  F = " << min_diag.number_of_faces() << std::endl;

  std::cout << "Construction took " << timer.time()
            << " seconds." << std::endl;

  return 0;
}

#endif
```

File: examples/Envelope_3/envelope_spheres.cpp

36.3.3 Example for Envelope of Planes

The next example demonstrates how to instantiate and use the envelope-traits class for planes, based on the *Arr_linear_traits_2* class that handles infinite linear objects such as lines and rays.

```
#include <CGAL/Gmpq.h>
#include <CGAL/Cartesian.h>
#include <CGAL/Env_plane_traits_3.h>
#include <CGAL/envelope_3.h>
#include <iostream>
#include <list>

typedef CGAL::Gmpq                                              Number_type;
typedef CGAL::Cartesian<Number_type>                            Kernel;
typedef Kernel::Plane_3                                         Plane_3;
```

```
typedef CGAL::Env_plane_traits_3<Kernel>                    Traits_3;
typedef Traits_3::Surface_3                                 Surface_3;
typedef CGAL::Envelope_diagram_2<Traits_3>
    Envelope_diagram_2;


/* Auxiliary function - print the features of the given envelope diagram.
     */
void print_diagram (const Envelope_diagram_2& diag)
{
  // Go over all arrangement faces.
  Envelope_diagram_2::Face_const_iterator            fit;
  Envelope_diagram_2::Ccb_halfedge_const_circulator  ccb;
  Envelope_diagram_2::Surface_const_iterator         sit;

  for (fit = diag.faces_begin(); fit != diag.faces_end(); ++fit)
  {
    // Print the face boundary.

    // Print the vertices along the outer boundary of the face.
    ccb = fit->outer_ccb();
    std::cout << "[Face]  ";
    do
    {
      if(!ccb->is_fictitious())
        std::cout << '(' << ccb->curve() << ") ";
      ++ccb;
    } while (ccb != fit->outer_ccb());

    // Print the planes that induce the envelope on this face.
    std::cout << "-->  " << fit->number_of_surfaces()
              << " planes:";

    for (sit = fit->surfaces_begin(); sit != fit->surfaces_end(); ++sit)
      std::cout << ' ' << sit->plane();
    std::cout << std::endl;
  }

  return;
}

/* The main program: */
int main ()
{
  // Construct the input planes.
  std::list<Surface_3>   planes;

  planes.push_back (Surface_3(Plane_3(0, -1, 1, 0)));
  planes.push_back (Surface_3(Plane_3(-1, 0, 1, 0)));
  planes.push_back (Surface_3(Plane_3(0, 1 , 1, 0)));
  planes.push_back (Surface_3(Plane_3(1, 0, 1,  0)));

  // Compute and print the minimization diagram.
  Envelope_diagram_2      min_diag;
```

```
  CGAL::lower_envelope_3 (planes.begin(), planes.end(), min_diag);

  std::cout << std::endl << "The minimization diagram:" << std::endl;
  print_diagram (min_diag);

  // Compute and print the maximization diagram.
  Envelope_diagram_2      max_diag;

  CGAL::upper_envelope_3 (planes.begin(), planes.end(), max_diag);

  std::cout << std::endl << "The maximization diagram:" << std::endl;
  print_diagram (max_diag);

  return (0);
}
```

File: examples/Envelope_3/envelope_planes.cpp

Envelopes of Surfaces in 3D Reference Manual

Dan Halperin, Michal Meyerovitch, Ron Wein and Baruch Zukerman

Introduction

This package consists of functions that compute the lower (or upper) envelope of a set of arbitrary surfaces in 3D. The output is represented as an 2D envelope diagram — a planar subdivision such that the identity of the surfaces that induce the envelope over each diagram cell is unique.

Functions

Concepts

Classes

36.4 Alphabetical List of Reference Pages

CGAL::lower_envelope_3

#include <CGAL/envelope_3.h>

template<class InputIterator, class Traits>
void lower_envelope_3(InputIterator begin, InputIterator end, Envelope_diagram_2<Traits>& diag)

Computes the lower envelope of a set of surfaces in $\mathbb{R}^3$, as given by the range *[begin, end)*. The lower envelope is represented using the output minimization diagram *diag*.
Precondition: The value-type of *InputIterator* is *Traits::Surface_3*.

Function

Function

CGAL::upper_envelope_3

#include <CGAL/envelope_3.h>

template<class InputIterator, class Traits>
void *upper_envelope_3(InputIterator begin, InputIterator end, Envelope_diagram_2<Traits>& diag)*

Computes the upper envelope of a set of surfaces in $\mathbb{R}^3$, as given by the range *[begin, end)*. The upper envelope is represented using the output maximization diagram *diag*.
Precondition: The value-type of *InputIterator* is *Traits::Surface_3*.

CGAL::lower_envelope_xy_monotone_3

#include <CGAL/envelope_3.h>

template<class InputIterator, class Traits>
void *lower_envelope_xy_monotone_3(InputIterator begin, InputIterator end, Envelope_diagram_2<Traits>& diag)*

Computes the lower envelope of a set of *xy*-monotone surfaces in $\mathbb{R}^3$, as given by the range *[begin, end)*. The lower envelope is represented using the output minimization diagram *diag*.
Precondition: The value-type of *InputIterator* is *Traits::Xy_monotone_surface_3*.

Function

Function

CGAL::upper_envelope_xy_monotone_3

#include <CGAL/envelope_3.h>

```
template<class InputIterator, class Traits>
void    upper_envelope_xy_monotone_3( InputIterator begin,
                                      InputIterator end,
                                      Envelope_diagram_2<Traits>& diag)
```

Computes the upper envelope of a set of *xy*-monotone surfaces in $\mathbb{R}^3$, as given by the range *[begin, end)*. The lower envelope is represented using the output maximization diagram *diag*.

Precondition: The value-type of *InputIterator* is *Traits::Xy_monotone_surface_3*.

EnvelopeTraits_3

Concept

Definition

This concept defines the minimal set of geometric predicates and operations needed to compute the envelope of a set of arbitrary surfaces in $\mathbb{R}^3$. It refines the *ArrangementXMonotoneTraits_2* concept. In addition to the *Point_2* and *X_monotone_curve_2* types and the *Has_boundary_category* category tag listed in the base concept, it also lists the *Surface_3* and *Xy_monotone_surface_3* types, which represent arbitrary surfaces and *xy*-monotone surfaces, respectively, and some constructions and predicates on these types. Note however, that these operations usually involve the projection of 3D objects onto the *xy*-plane.

Refines

ArrangementXMonotoneTraits_2

Types

EnvelopeTraits_3:: Surface_3	represents an arbitrary surface in $\mathbb{R}^3$.
EnvelopeTraits_3:: Xy_monotone_surface_3	represents a weakly *xy*-monotone surface in $\mathbb{R}^3$.

Functor Types

EnvelopeTraits_3:: Make_xy_monotone_3

provides the operator (templated by the *OutputIterator* type) :

- *OutputIterator operator() (Surface_3 S, bool is_lower, OutputIterator oi)*
  which subdivides the given surface *S* into *xy*-monotone parts and inserts them into the output iterator. The value of *is_lower* indicates whether we compute the lower or the upper envelope, so that *xy*-monotone surfaces that are irrelevant to the lower-envelope (resp. upper-envelope) computation may be discarded. The value-type of *OutputIterator* is *Xy_monotone_surface_3*. The operator returns a past-the-end iterator for the output sequence.

EnvelopeTraits_3:: Construct_projected_boundary_2

provides the operator (templated by the *OutputIterator* type) :

- *OutputIterator operator() (Xy_monotone_surface_3 s, OutputIterator oi)*
  which computes all planar *x*-monotone curves and possibly isolated planar points that form the projection of the boundary of the given *xy*-monotone surface *s* onto the *xy*-plane, and inserts them into the output iterator. The value-type of *OutputIterator* is *Object*, where *Object* wraps either a *Point_2*, or a *pair<X_monotone_curve_2, Oriented_side>*. In the former case, the object represents an isolated point of the projected boundary. In the latter, more general, case the object represents an *x*-monotone boundary curve along with an enumeration value which is either *ON_NEGATIVE_SIDE* or *ON_POSITIVE_SIDE*, indicating whether whether the projection of the surface onto the *xy*-plane lies below or above this *x*-monotone curve, respectively. In degenerate case, namely when the surface itself is vertical, and its projection onto the plane is 1-dimensional, the *Oriented_side* value is *ON_ORIENTED_BOUNDARY*. The operator returns a past-the-end iterator for the output sequence.

EnvelopeTraits_3:: Construct_projected_intersections_2

provides the operator (templated by the *OutputIterator* type) :

- *OutputIterator operator() (Xy_monotone_surface_3 s1, Xy_monotone_surface_3 s2, OutputIterator oi)*
  which computes the projection of the intersections of the *xy*-monotone surfaces *s1* and *s2* onto the *xy*-plane, and inserts them into the output iterator. The value-type of *OutputIterator* is *Object*, where each *Object* either wraps a *pair<X_monotone_curve_2,Multiplicity>* instance, which represents a projected intersection curve with its multiplicity (in case the multiplicity is undefined or not known, it should be set to 0) or an *Point_2* instance, representing the projected image of a degenerate intersection (the projection of an isolated intersection point, or of a vertical intersection curve). The operator returns a past-the-end iterator for the output sequence.

EnvelopeTraits_3:: Compare_z_at_xy_3

provides the operators :

- *Comparison_result operator() (Point_2 p, Xy_monotone_surface_3 s1, Xy_monotone_surface_3 s2)*
  which determines the relative z-order of the two given *xy*-monotone surfaces at the *xy*-coordinates of the point p, with the precondition that both surfaces are defined over p. Namely, it returns the comparison result of $s_1(p)$ and $s_2(p)$.

- *Comparison_result operator() (X_monotone_curve_2 c, Xy_monotone_surface_3 s1, Xy_monotone_surface_3 s2)*
  which determines the relative z-order of the two given *xy*-monotone surfaces over the interior of a given x-monotone curve c, with the precondition that c is fully contained in the *xy*-definition range of both s_1 and s_2, and that the surfaces do not intersect over c. The functor should therefore return the comparison result of $s_1(p')$ and $s_2(p')$ for some point p' in the interior of c.

- *Comparison_result operator() (Xy_monotone_surface_3 s1, Xy_monotone_surface_3 s2)*
  which determines the relative z-order of the two given unbounded *xy*-monotone surfaces, which are defined over the entire *xy*-plane and have no boundary, with the precondition that the surfaces do not intersect at all. The functor should therefore return the comparison result of $s_1(p)$ and $s_2(p)$ for some planar point $p \in \mathbb{R}^2$. This operator is required iff the category tag *Has_boundary_category* is defined as *Tag_true*.

EnvelopeTraits_3:: Compare_z_at_xy_above_3

provides the operator :

- *Comparison_result operator() (X_monotone_curve_2 c, Xy_monotone_surface_3 s1, Xy_monotone_surface_3 s2)*
  which determines the relative z-order of the two given *xy*-monotone surfaces immediately above their projected intersection curve c (a planar point p is *above* an x-monotone curve c if it is in the x-range of c, and lies to its left when the curve is traversed from its *xy*-lexicographically smaller endpoint to its larger endpoint). We have the precondition that both surfaces are defined "above" c, and their relative z-order is the same for some small enough neighborhood of points above c.

EnvelopeTraits_3:: Compare_z_at_xy_below_3

provides the operator :

- *Comparison_result operator() (X_monotone_curve_2 c, Xy_monotone_surface_3 s1, Xy_monotone_surface_3 s2)*
  which determines the relative z-order of the two given xy-monotone surfaces immediately below their projected intersection curve c (a planar point p is *below* an x-monotone curve c if it is in the x-range of c, and lies to its right when the curve is traversed from its xy-lexicographically smaller endpoint to its larger endpoint). We have the precondition that both surfaces are defined "below" c, and their relative z-order is the same for some small enough neighborhood of points below c.

Creation

EnvelopeTraits_3	*traits*;	default constructor.
EnvelopeTraits_3	*traits(other);*	copy constructor.
EnvelopeTraits_3	*traits = other*	assignment operator.

Accessing Functor Objects

Make_xy_monotone_3	*traits.make_xy_monotone_3_object()*
Construct_projected_boundary_2	*traits.construct_projected_boundary_2_object()*
Construct_projected_intersections_2	*traits.construct_projected_intersections_2_object()*
Compare_z_at_xy_3	*traits.compare_z_at_xy_3_object()*
Compare_z_at_xy_above_3	*traits.compare_z_at_xy_above_3_object()*
Compare_z_at_xy_below_3	*traits.compare_z_at_xy_below_3_object()*

Has Models

CGAL::Env_triangle_traits_3<Kernel>
CGAL::Env_sphere_traits_3<ConicTraits>
CGAL::Env_plane_traits_3<Kernel>
CGAL::Env_surface_data_traits_3<Traits,XyData,SData,Cnv>

Class

CGAL::Env_triangle_traits_3<Kernel>

Definition

The traits class *Env_triangle_traits_3<Kernel>* models the *EnvelopeTraits_3* concept, and is used for the construction of lower and upper envelopes of triangles in the space. It is parameterized by a CGAL-kernel, which is parameterized in turn by a number type. The number type should support exact rational arithmetic, to avoid numerical errors and robustness problems. In particular, the number type should support the arithmetic operations $+$, $-$, $\times$, and $\div$ without loss of precision. For optimal performance, we recommend instantiating the traits class with the predefined *Exact_predicates_exact_constructions_kernel* provided by CGAL. Using this kernel guarantees exactness and robustness, while it incurs only a minor overhead (in comparison to working with a fast, inexact number type) for most inputs.

Note that when we project the boundary of a triangle, or the intersection of two triangles, onto the *xy*-plane, we obtain line segments. Indeed, *Env_triangle_traits_3<Kernel>* inherits from the *Arr_segment_traits_2<Kernel>* traits class, and extends it by adding operations on 3D objects, namely spacial triangles. Note that the traits class does *not* define *Kernel::Triangle_3* as its surface (and *xy*-monotone surface) type, as one may expect. This is because the traits class needs to store extra data with the triangles in order to efficiently operate on them. Nevertheless, the nested *Xy_monotone_surface_3* and *Surface_3* types are however constructible from a *Kernel::Triangle_3* instance and are also convertible to a *Kernel::Triangle_3* object. Both types, *Xy_monotone_surface_3* and *Surface_3*, refer to the same class, as *every* triangle is (weakly) *xy*-monotone).

#include <CGAL/Env_triangle_traits_3.h>

Is Model for the Concepts

EnvelopeTraits_3

Inherits From

Arr_segment_traits_2<Kernel>

CGAL::Env_sphere_traits_3<ConicTraits>

Class

Definition

The traits class *Env_sphere_traits_3<ConicTraits>* models the *EnvelopeTraits_3* concept, and is used for the construction of lower and upper envelopes of spheres. Note that when projecting the intersection curve of two spheres (a circle in 3D) onto the *xy*-plane, the resulting curve is an ellipse. The traits class is therefore parameterized by an arrangement-traits class that is capable of handling conic curves — namely an instantiation of the *Arr_conic_traits_2* class-template — and inherits from it.

The conic-traits class defines a nested type named *Rat_kernel*, which is a geometric kernel parameterized by an exact rational type. *Env_sphere_traits_3<ConicTraits>* defines its *Surface_3* type to be constructible from *Rat_kernel::Sphere_3*. Namely, it can handle spheres whose center points have rational coordinates (i.e., of the type *Rat_kernel::FT*), and whose squared radius is also rational. The *Surface_3* type is also convertible to a *Rat_kernel::Sphere_3* object.

The *Xy_monotone_surface_3* type is the same as the nested *Surface_3* type. The traits-class simply ignores the upper hemisphere when it computes lower envelopes, and ignores the lower hemisphere when it computes upper envelopes.

#include <CGAL/Env_sphere_traits_3.h>

Is Model for the Concepts

EnvelopeTraits_3

Inherits From

ConicTraits

Class

CGAL::Env_surface_data_traits_3<Traits,XyData,SData,Cnv>

Definition

The class *Env_surface_data_traits_3<Traits,XyData,SData,Cnv>* is a model of the *EnvelopeTraits_3* concept and serves as a decorator class that allows the extension of the surfaces defined by the base traits-class (the *Traits* parameter), which serves as a geometric traits-class (a model of the *EnvelopeTraits_3* concept), with extraneous (non-geometric) data fields.

The traits-class decorator extends the *Surface_3* and the *Xy_monotone_surface_3* types as detailed below. Each *Surface_3* object is associated with a single data field of type *SData*, and each *Xy_monotone_surface_3* object is associated with a single data field of type *XyData*. When a surface is subdivided into *xy*-monotone surfaces, its data field is converted using the conversion functor, which is specified by the *Cnv* template-parameter, and the resulting objects is copied to all *Xy_monotone_surface_3* objects induced by this surface. The conversion functor should provide an operator with the following prototype:
XyData operator() (const SData& d) const;

By default, the two data types are the same, so the conversion operator is trivial and just copies the data object:

SData = XyData
Cnv = _Default_convert_functor<SData,XyData>

#include <CGAL/Env_surface_data_traits_3.h>

Is Model for the Concepts

EnvelopeTraits_3

Types

typedef Traits	*Base_traits_3;*	the base traits-class.
typedef typename Base_traits_3::Surface_3	*Base_surface_3;*	the base surface.
typedef typename Base_traits_3::Xy_monotone_surface_3	*Base_xy_monotone_surface_3;*	the base *xy*-monotone surface surface.
typedef Cnv	*Convert;*	the conversion functor.
typedef SData	*Surface_data;*	the type of data associated with surfaces.
typedef XyData	*Xy_monotone_surface_data;*	the type of data associated with *xy*-monotone surfaces.

Inherits From

Base_traits_3

Class Env_surface_data_traits_3<Traits,XyData,SData,Cnv>::Surface_3

The *Surface_3* class nested within the surface-data traits extends the *Base_traits_3::Surface_3* type with an extra data field.

Inherits From

Base_surface_3

Creation

Env_surface_data_traits_3<Traits,XyData,SData,Cnv>::Surface_3 s;

default constructor.

Env_surface_data_traits_3<Traits,XyData,SData,Cnv>::Surface_3 s(Base_surface_3 base);

constructs surface from the given *base* surface with uninitialized data field.

Env_surface_data_traits_3<Traits,XyData,SData,Cnv>::Surface_3 s(Base_surface_3 base, Surface_data data)

constructs surface from the given *base* surface with an attached *data* field.

Access Functions

Surface_data *s.data() const* returns the data field (a non-const version, which returns a reference to the data object, is also available).

void *s.set_data(Surface_data data)*

sets the data field.

Class Env_surface_data_traits_3<Traits,XyData,SData,Cnv>::Xy_monotone_surface_3

The *Xy_monotone_surface_3* class nested within the surface-data traits extends the *Base_traits_3::Xy_monotone_surface_3* type with an extra data field.

Inherits From

Base_xy_monotone_surface_3

Creation

Arr_surface_data_traits_3<Tr,XData,Mrg,CData,Cnv>::Xy_monotone_surface_3 xys;

default constructor.

Arr_surface_data_traits_3<Tr,XData,Mrg,CData,Cnv>::Xy_monotone_surface_3 xys(Base_xy_monotone_surface_3 base)

constructs an *xy*-monotone surface from the given *base* surface with uninitialized data field.

Arr_surface_data_traits_3<Tr,XData,Mrg,CData,Cnv>::Xy_monotone_surface_3 xys(Base_xy_monotone_surface_3 base, Xy_monotone_surface_data data)

constructs an *x*-monotone surface from the given *base* *x*-monotone surface with an attached *data* field.

Access Functions

Xy_monotone_surface_data *xys.data() const*

returns the field (a non-const version, which returns a reference to the data object, is also available).

void *xys.set_data(Xy_monotone_surface_data data)*

sets the data field.

CGAL::Env_plane_traits_3<Kernel>

Definition

The traits class *Env_plane_traits_3<Kernel>* models the *EnvelopeTraits_3* concept, and is used for the construction of lower and upper envelopes of planes and half planes in the space. It is parameterized by a CGAL-kernel model, which is parameterized in turn by a number type. The number type should support exact rational arithmetic, to avoid numerical errors and robustness problems. In particular, the number type should support the arithmetic operations $+$, $-$, $\times$, and $\div$ without loss of precision. For optimal performance, we recommend instantiating the traits class with the predefined *Exact_predicates_exact_constructions_kernel* provided by CGAL. Using this kernel guarantees exactness and robustness, while it incurs only a minor overhead (in comparison to working with a fast, inexact number type) for most inputs.

Note that an entire plane has no boundaries, and the projection of a half-plane is an (unbounded) line. Naturally, rays and segments may occur as a result of overlaying projections of several half planes. Indeed, *Env_plane_traits_3<Kernel>* inherits from the *Arr_linear_traits_2<Kernel>* traits class, and extends it by adding operations on planes and half planes. The nested *Xy_monotone_surface_3* and *Surface_3* types refer to the same type. They are constructible from a *Kernel::Plane_3* in case of an entire plane, or from *Kernel::Plane_3* and *Kernel::Line_2* in case of a half-plane. The line orientation determines which half is considered.

#include <CGAL/Env_plane_traits_3.h>

Is Model for the Concepts

EnvelopeTraits_3

Inherits From

Arr_linear_traits_2<Kernel>

CGAL::Envelope_diagram_2<EnvTraits>

Definition

The class-template *Envelope_diagram_2<EnvTraits>* represents the minimization diagram that corresponds to the lower envelope of a set of curves, or the maximization diagram that corresponds to their upper envelope. It is parameterized by a traits class that must be a model of the *EnvelopeTraits_3* concept, and is basically a planar arrangement of x-monotone curves, as defined by this traits class. These x-monotone curves are the projections of boundary curves of xy-monotone surfaces, or the intersection curves between such surfaces, onto the xy-plane. Thus, it is possible to traverse the envelope diagram using the methods inherited from the *Arrangement_2* class.

The envelope diagram extends the arrangement features (namely the vertices, halfedges, and faces), such that each feature stores a container of originators — namely, the xy-monotone surfaces (instances of the type *EnvTraits::Xy_monotone_surface_3*) that induce the lower envelope (or the upper envelope, in case of a maximization diagram) over this feature. The envelope diagram provides access methods to these originators.

#include <CGAL/envelope_3.h>

Inherits From

Arrangement_2<EnvTraits>

Types

typedef Envelope_diagram_2<EnvTraits>

Self;

typedef Arrangement_2<EnvTraits>

Base;

Envelope_diagram_2<EnvTraits>:: Surface_const_iterator

an iterator for the xy-monotone surfaces that induce a diagram feature. Its value-type is *EnvTraits::Xy_monotone_surface_3*.

Creation

Envelope_diagram_2<EnvTraits> diag; — constructs an empty diagram containing one unbounded face, which corresponds to the entire plane and has no originators.

Envelope_diagram_2<EnvTraits> diag(Self other);

copy constructor.

*Envelope_diagram_2<EnvTraits> diag(EnvTraits *traits);*

constructs an empty diagram that uses the given *traits* instance for performing the geometric predicates.

Class Envelope_diagram_2<EnvTraits>::Vertex

Inherits From

Base::Vertex

Access Functions

size_t *v.number_of_surfaces() const*

returns the number of *xy*-monotone surfaces that induce *v*.

Surface_const_iterator

v.surfaces_begin() const

returns an iterator for the first *xy*-monotone surface that induces *v*.

Surface_const_iterator

v.surfaces_end() const

returns a past-the-end iterator for the *xy*-monotone surfaces that induce *v*.

Xy_monotone_surface_3

v.surface() const returns the first *xy*-monotone surface that induce *v*.
Precondition: The number of surfaces is not 0.

Class Envelope_diagram_2<EnvTraits>::Halfedge

Inherits From

Base::Halfedge

Access Functions

size_t *e.number_of_surfaces() const*

returns the number of *xy*-monotone surfaces that induce *e*.

Surface_const_iterator

e.surfaces_begin() const

returns an iterator for the first *xy*-monotone surface that induces *e*.

Surface_const_iterator

e.surfaces_end() const

returns a past-the-end iterator for the *xy*-monotone surfaces that induce *e*.

Xy_monotone_surface_3

e.surface() const returns the first *xy*-monotone surface that induce *e*.
Precondition: The number of surfaces is not 0.

Class Envelope_diagram_2<EnvTraits>::Face

Inherits From

Base::Face

Access Functions

size_t *f.number_of_surfaces() const*

returns the number of *xy*-monotone surfaces that induce *f*.

Surface_const_iterator

f.surfaces_begin() const

returns an iterator for the first *xy*-monotone surface that induces *f*.

Surface_const_iterator

f.surfaces_end() const

returns a past-the-end iterator for the *xy*-monotone surfaces that induce *f*.

Xy_monotone_surface_3

f.surface() const returns the first *xy*-monotone surface that induce *f*.
Precondition: The number of surfaces is not 0.

Part IX

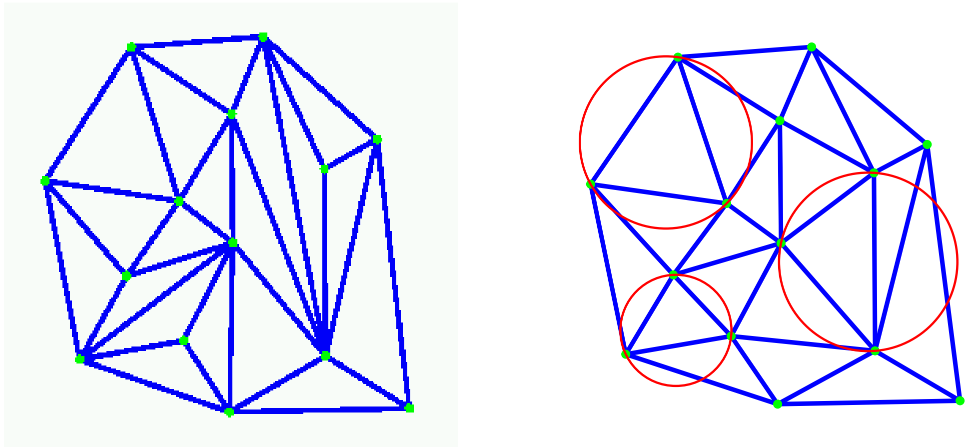
Triangulations and Delaunay Triangulations

Chapter 37

2D Triangulations

Mariette Yvinec

Contents



This chapter describes the two dimensional triangulations of CGAL. Section 37.1 recalls the main definitions about triangulations. Sections 37.2 discusses the way two-dimensional triangulations are represented in CGAL. Section 37.3 presents the overall software design of the 2D triangulations package. The next sections present the different two dimensional triangulations classes available in CGAL : basic triangulations (section 37.4), Delaunay triangulations (Section 37.5), regular triangulations (Section 37.6), constrained triangulations (Section 37.7), and constrained Delaunay triangulations (Section 37.8). Section 37.9 describes a class which implements a constrained or constrained Delaunay triangulation with an additional data structure to describe how the constraints are refined by the edges of the triangulations. Section 37.10 describes a hierarchical data structure for fast point location queries. At last, Section 37.11 explains how the user can benefit from the flexibility of CGAL triangulations using customized classes for faces and vertices.

37.1 Definitions

A two dimensional triangulation can be roughly described as a set T of triangular facets such that :
- two facets either are disjoint or share a lower dimensional face (edge or vertex).
- the set of facets in T is connected for the adjacency relation.
- the domain U_T which is the union of facets in T has no singularity.

More precisely, a triangulation can be described as a simplicial complex. Let us first record a few definitions.

A simplicial complex is a set T of simplices such that
- any face of a simplex in T is a simplex in T
- two simplices in T either are disjoint or share a common sub-face.

The dimension d of a simplicial complex is the maximal dimension of its simplices.
A simplicial complex T is pure if any simplex of T is included in a simplex of T with maximal dimension.
Two simplexes in T with maximal dimension d are said to be adjacent if they share a $d-1$ dimensional sub-face.
A simplicial complex is connected if the adjacency relation defines a connected graph over the set of simplices of T with maximal dimension.
The union U_T of all simplices in T is called the domain of T. A point p in the domain of T is said to singular if its surrounding in U_T is neither a topological ball nor a topological disc.

Then, a two dimensional triangulation can be described as a two dimensional simplicial complex that is pure, connected and without singularity.

Each facet of a triangulation can be given an orientation which in turn induces an orientation on the edges incident to that facet. The orientation of two adjacent facets are said to be consistent if they induce opposite orientations on their common incident edge. A triangulation is said to be orientable if the orientation of each facet can be chosen in such a way that all pairs of incident facets have consistent orientations.

The data structure underlying CGAL triangulations allows to represent the combinatorics of any orientable two dimensional triangulations without boundaries. On top of this data structure, the 2D triangulations classes take care of the geometric embedding of the triangulation and are designed to handle planar triangulations. The plane of the triangulation may be embedded in a higher dimensional space.

The triangulations of CGAL are complete triangulations which means that their domain is the convex hull of their vertices. Because any planar triangulation can be completed, this is not a real restriction. For instance, a triangulation of a polygonal region can be constructed and represented as a subset of a constrained triangulation in which the region boundary edges have been input as constrained edges (see Section 37.7, 37.8 and 37.9).

Strictly speaking, the term *face* should be used to design a face of any dimension, and the two-dimensional faces of a triangulation should be properly called *facets*. However, following a common usage, we hereafter often call *faces*, the facets of a two dimensional triangulation.

37.2 Representation

The Set of Faces

A 2D triangulation of CGAL can be viewed as a planar partition whose bounded faces are triangular and cover the convex hull of the set of vertices. The single unbounded face of this partition is the complementary of the convex hull. In many applications, such as Kirkpatrick's hierarchy or incremental Delaunay construction, it is convenient to deal with only triangular faces. Therefore, a fictitious vertex, called the *infinite vertex* is added to the triangulation as well as *infinite edges* and *infinite faces* incident to it.. Each infinite edge is incident to the infinite vertex and to a vertex of the convex hull. Each infinite face is incident to the infinite vertex and to a convex hull edge.

Therefore, each edge of the triangulation is incident to exactly two faces and the set of faces of a triangulation is topologically equivalent to a two-dimensional sphere. This extends to lower dimensional triangulations arising in degenerate cases or when the triangulations as less than three vertices. Including the infinite faces, a one dimensional triangulation is a ring of edges and vertices topologically equivalent to a 1-sphere. A zero dimensional triangulation, whose domain is reduced to a single point, is represented by two vertices that is topologically equivalent to a 0-sphere.

Note that the *infinite vertex* has no significant coordinates and that no geometric predicate can be applied on it nor on an infinite face.

A Representation Based on Faces and Vertices

Because a triangulation is a set of triangular faces with constant-size complexity, triangulations are not implemented as a layer on top of a planar map. CGAL uses a proper internal representation of triangulations based on faces and vertices rather than on edges. Such a representation saves storage space and results in faster algorithms [BDTY00].

The basic elements of the representation are vertices and faces. Each triangular face gives access to its three incident vertices and to its three adjacent faces. Each vertex gives access to one of its incident faces and through that face to the circular list of its incident faces.

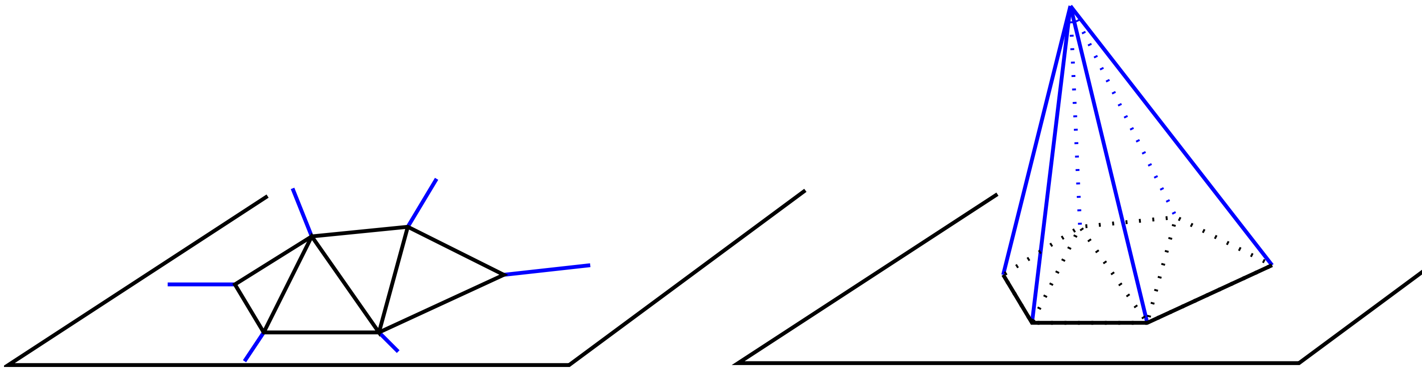


Figure 37.1: Infinite vertex and infinite faces

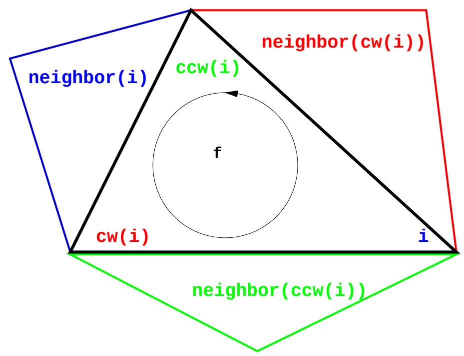


Figure 37.2: Vertices and neighbors.

The three vertices of a face are indexed with 0, 1 and 2 in counterclockwise order. The neighbors of a face are also indexed with 0,1,2 in such a way that the neighbor indexed by i is opposite to the vertex with the same index. See Figure 37.2, the functions *ccw(i)* and *cw(i)* shown on this figure compute respectively $i+1$ and $i-1$ modulo 3.

The edges are not explicitly represented, they are only implicitly represented through the adjacency relations of two faces. Each edge has two implicit representations: the edge of a face f which is opposed to the vertex indexed i, can be represented as well as an edge of the *neighbor(i)* of f.

37.3 Software Design

The triangulations classes of CGAL provide high-level geometric functionalities such as location of a point in the triangulation, insertion, removal, or displacement of a point. They are build as a layer on top of a data structure called the triangulation data structure. The triangulation data structure can be thought of as a container for the faces and vertices of the triangulation. This data structure also takes care of all the combinatorial aspects of the triangulation.

This separation between the geometric aspect and the combinatorial part is reflected in the software design by the fact that the triangulation classes have two template parameters:

- the first parameter stands for a **geometric traits** class providing the geometric primitives (points, segments

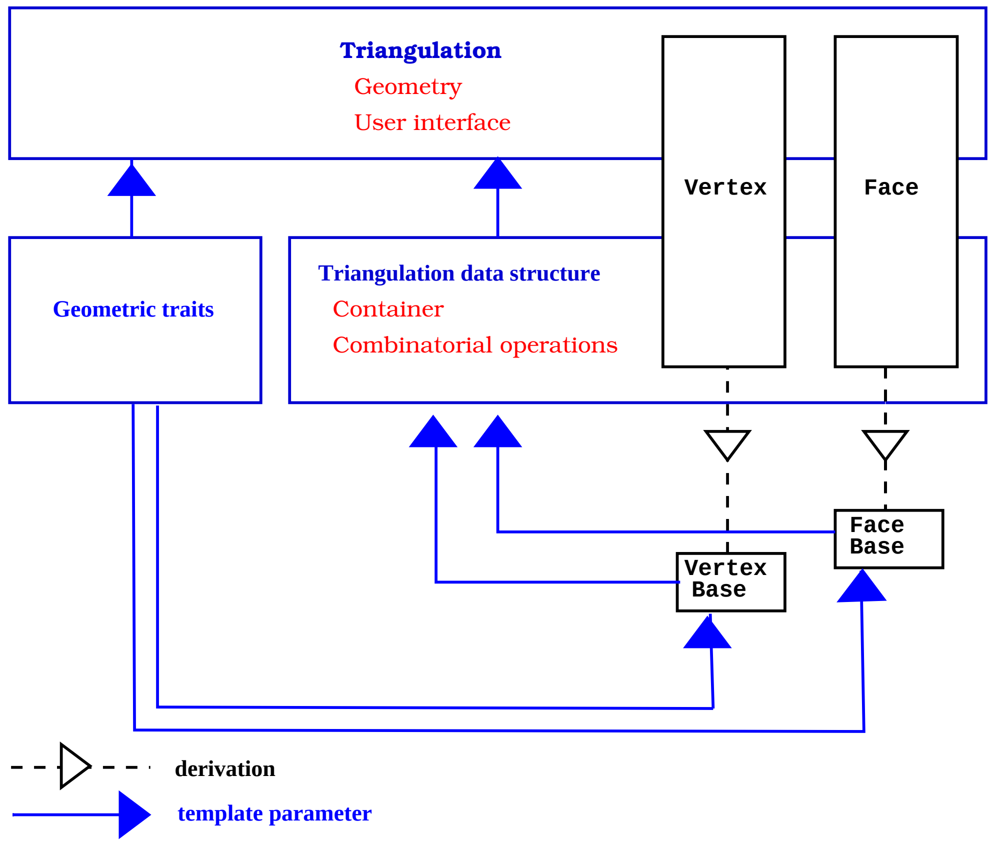


Figure 37.3: The triangulations software design.

and triangles) of the triangulation and the elementary operations (predicate or constructions) on those objects.

- the second parameter stands for a **triangulation data structure** class. The concept of triangulation data structure is described in Section 38.2 of Chapter 38. The triangulation data structure defines the types used to represent the faces and vertices of the triangulation, as well as additional types (handles, iterators and circulators) to access and visit the faces and vertices.

  CGAL provides the class *Triangulation_data_structure_2<Vb,Fb>* as a default model of triangulation data structure. The class *Triangulation_data_structure_2<Vb,Fb>* has two template parameters standing for a vertex class and a face class. CGAL defines concepts for these template parameters and provide default models for these concepts. The vertex and base classes are templated by the geometric traits which allows them to have some knowledge of the geometric primitives of the triangulation. Those default vertex and face base classes can be replaced by user customized base classes in order, for example, to deal with additional properties attached to the vertices or faces of a triangulation. See section 37.11 for more details on the way to make use of this flexibility.

The Figure 37.3 summarizes the design of the triangulation package, showing the three layers (base classes, triangulation data structure and triangulation) forming this design.

The top triangulation level, responsible for the geometric embedding of the triangulation comes in different flavors according to the different kind of triangulations: basic, Delaunay, regular, constrained or constrained Delaunay. Each kind of triangulations correspond to a different class. Figure 37.4 summarizes the derivation dependencies of CGAL 2D triangulations classes. Any 2D triangulation class is parametrized by a geometric

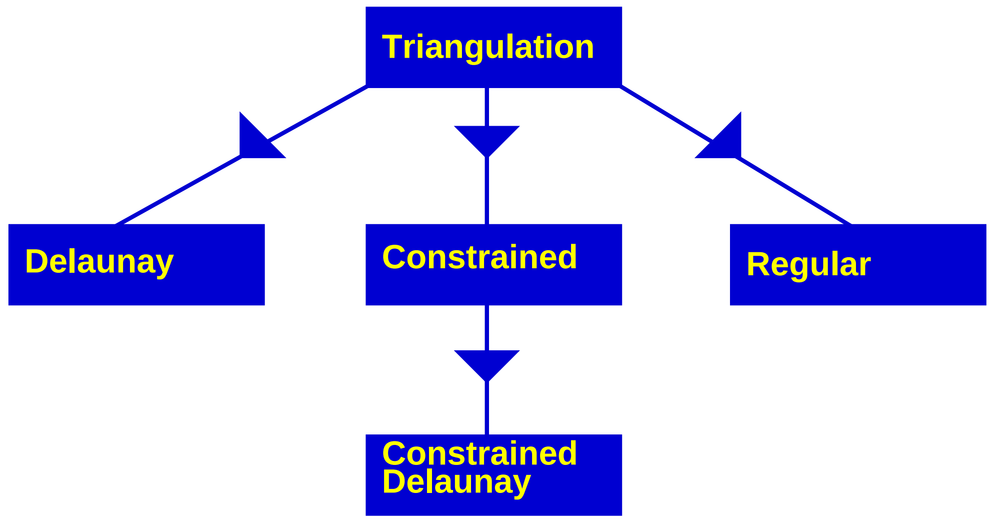


Figure 37.4: The derivation tree of 2D triangulations.

traits and a triangulation data structure. While a unique concept *TriangulationDataStructure_2* describes the triangulation data structure requirements for any triangulation class, the concept of geometric traits actually depends on the triangulation class. In general, the requirements for the vertex and face base classes are described by the basic concepts *TriangulationVertexBase_2* and *TriangulationFaceBase_2*. However, some triangulation classes requires base classes implementing refinements of the basic concepts.

37.4 Basic Triangulations

37.4.1 Description

The class *Triangulation_2<Traits,Tds>* serves as a base class for the other 2D triangulations classes and implements the user interface to a triangulation.

The vertices and faces of the triangulations are accessed through *handles*, *iterators* and *circulators*. A handle is a model of the concept *Handle* which basically offers the two dereference operators * and -> . A circulator is a type devoted to visit circular sequences. Handles are used whenever the accessed element is not part of a sequence. Iterators and circulators are used to visit all or parts of the triangulation.

The iterators and circulators are all bidirectional and non mutable. The circulators and iterators are convertible to the handles with the same value type, so that when calling a member function, any handle type argument can be replaced by an iterator or a circulator with the same value type.

The triangulation class allows to visit the vertices and neighbors of a face in clockwise or counterclockwise order.

There are circulators to visit the edges or faces incident to a given vertex or the vertices adjacent to it. Another circulator type allows to visit all the faces traversed by a given line. Circulators step through infinite features as well as through finite ones.

The triangulation class offers some iterators to visit all the faces, edges or vertices and also iterators to visit selectively the finite faces, edges or vertices.

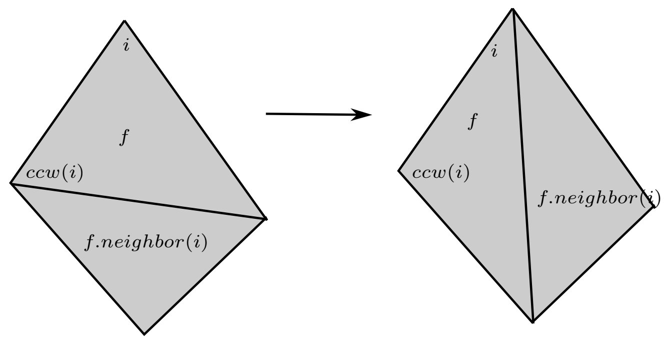


Figure 37.5: Flip.

The triangulation class provides methods to test the infinite character of any feature, and also methods to test the presence in the triangulation of a particular feature (edge or face) given its vertices.

The triangulation class provides a method to locate a given point with respect to a triangulation. In particular, this method reports whether the point coincides with a vertex of the triangulation, lies on an edge, in a face or outside of the convex hull. In case of a degenerate lower dimensional triangulation, the query point may also lie outside the triangulation affine hull.

The triangulation class also provides methods to locate a point with respect to a given finite face of the triangulation or with respect to its circumcircle. The faces of the triangulation and their circumcircles have the counterclockwise orientation.

The triangulation can be modified by several functions: insertion of a point, removal of a vertex, displacement of a vertex, flipping of an edge. The flipping of an edge is possible when the union of the two incident faces forms a convex quadrilateral (see Figure 37.5).

Implementation

Locate is implemented by a line walk. The walk begins at a vertex of the face which is given as an optional argument or at an arbitrary vertex of the triangulation if no optional argument is given. It takes time $O(n)$ in the worst case, but only $O(\sqrt{n})$ on average if the vertices are distributed uniformly at random. The class *Triangulation_hierarchy_2<Traits,Tds>*, described in section 37.10, implements a data structure designed to offer an alternate more efficient point location algorithm.

Insertion of a point is done by locating a face that contains the point, and splitting this face into three new faces. If the point falls outside the convex hull, the triangulation is restored by flips. Apart from the location, insertion takes a time $O(1)$. This bound is only an amortized bound for points located outside the convex hull.

Removal of a vertex is done by removing all adjacent triangles, and re-triangulating the hole. Removal takes a time at most proportional to d^2, where d is the degree of the removed vertex, which is $O(1)$ for a random vertex.

Displacement of a vertex is done by: first, verifying if the triangulation embedding remains planar after the displacement; if yes the vertex is directly placed at the new location; otherwise, a point is inserted at the new location and the vertex at the obsolete location is removed.

The face, edge, and vertex iterators on finite features are derived from their counterparts visiting all (finite and infinite) features which are themselves derived from the corresponding iterators of the triangulation data structure.

Geometric Traits

The geometric traits of a triangulation is required to provide the geometric objects (points, segments and triangles) building up the triangulation together with the geometric predicates on those objects. The required predicates are:
- comparison of the x or y coordinates of two points.
- the orientation test which computes the order type of three given point.

The concept *TriangulationTraits_2* describes the requirements for the geometric traits class of a triangulation. The CGAL kernel classes are models for this concept. The CGAL library also provides dedicated models of *TriangulationTraits_2* using the kernel geometric objects and predicates. These classes are themselves templated with a CGAL kernel and extract the required types and predicates from the kernel. The traits class *Triangulation_euclidean_traits_2<R>* is designed to deal with ordinary two dimensional points. The class *Projection_traits_xy_3<R>* is a geometric traits class to build the triangulation of a terrain. Such a triangulation is a two-dimensional triangulation embedded in three dimensional space. The data points are three-dimensional points. The triangulation is build according to the projections of those points on the xy plane and then lifted up to the original three-dimensional data points. This is especially useful to deal with GIS terrains. Instead of really projecting the three-dimensional points and maintaining a mapping between each point and its projection (which costs space and is error prone), the traits class supplies geometric predicates that ignore the z-coordinates of the points. See Section 37.5 for an example. CGAL provides also the geometric traits classes *Projection_traits_yz_3<R>* and *Projection_traits_xz_3<R>* to deal with projections on the yz plane and xz-plane, respectively.

37.4.2 Example of a Basic Triangulation

The following program creates a triangulation of 2D points using the default kernel *Exact_predicate_inexact_constructions_kernel* as geometric traits and the default triangulation data structure. The input points are read from a file and inserted in the triangulation. Finally points on the convex hull are written to `cout`.

```
#include <fstream>

#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/Triangulation_2.h>

typedef CGAL::Exact_predicates_inexact_constructions_kernel K;

typedef CGAL::Triangulation_2<K>         Triangulation;
typedef Triangulation::Vertex_circulator Vertex_circulator;
typedef Triangulation::Point             Point;

int main() {
  std::ifstream in("data/triangulation_prog1.cin");
  std::istream_iterator<Point> begin(in);
  std::istream_iterator<Point> end;

  Triangulation t;
  t.insert(begin, end);

  Vertex_circulator vc = t.incident_vertices(t.infinite_vertex()),
    done(vc);
  if (vc != 0) {
    do { std::cout << vc->point() << std::endl;
    }while(++vc != done);
```

```
  }
  return 0;
}
```

File: examples/Triangulation_2/triangulation_prog1.cpp

37.5 Delaunay Triangulations

37.5.1 Description

The class *Delaunay_triangulation_2<Traits,Tds>* is designed to represent the Delaunay triangulation of a set of data points in the plane. A Delaunay triangulation fulfills the following *empty circle property* (also called *Delaunay property*): the circumscribing circle of any facet of the triangulation contains no data point in its interior. For a point set with no subset of four co-circular points the Delaunay triangulation is unique, it is dual to the Voronoi diagram of the set of points.

The class *Delaunay_triangulation_2<Traits,Tds>* derives from the class *Triangulation_2<Traits,Tds>*.

The class *Delaunay_triangulation_2<Traits,Tds>* inherits the types defined by the basic class *Triangulation_2< Traits,Tds>*. Additional types, provided by the traits class, are defined to represent the dual Voronoi diagram.

The class *Delaunay_triangulation_2<Traits,Tds>* overwrites the member functions that insert, move, or remove a point in the triangulation to maintain the Delaunay property. It also has a member function (*Vertex_handle nearest_vertex(const Point& p)*) to answer nearest neighbor queries and member functions to construct the elements (vertices and edges) of the dual Voronoi diagram.

Geometric traits

The geometric traits has to be a model of the concept *DelaunayTriangulationTraits_2* which refines the concept *TriangulationTraits_2*. In particular this concept provides the *side_of_oriented_circle* predicate which, given four points *p,q,r,s* decides the position of the point *s* with respect to the circle passing through *p*, *q* and *r*. The *side_of_oriented_circle* predicate actually defines the Delaunay triangulation. Changing this predicate allows to build variant of Delaunay triangulations for different metrics such that L_1 or L_∞ metric or any metric defined by a convex object. However, the user of an exotic metric must be careful that the constructed triangulation has to be a triangulation of the convex hull which means that convex hull edges have to be Delaunay edges. This is granted for any smooth convex metric (like L_2) and can be ensured for other metrics (like L_∞) by the addition to the point set of well chosen sentinel points.

The CGAL kernel classes and the class *Triangulation_euclidean_traits_2<R>* are models of the concept *DelaunayTriangulationTraits_2* for the euclidean metric. The traits class for terrains, *Projection_traits_xy_3<R>*,
Projection_traits_yz_3<R>, and
Projection_traits_xz_3<R>
are also models of *DelaunayTriangulationTraits_2* except that they do not fulfill the requirements for the duality functions and nearest vertex queries.

Implementation

The insertion of a new point in the Delaunay triangulation is performed using first the insertion member function of the basic triangulation and second performing a sequence of flips to restore the Delaunay property. The number of flips that have to be performed is $O(d)$ if the new vertex has degree d in the updated Delaunay triangulation. For points distributed uniformly at random, each insertion takes time $O(1)$ on average, once the point has been located in the triangulation.

Removal calls the removal in the triangulation and then re-triangulates the hole created in such a way that the Delaunay criterion is satisfied. Removal of a vertex of degree d takes time $O(d^2)$. The degree d is $O(1)$ for a random vertex in the triangulation. When the degree of the removed vertex is small (≤ 7) a special procedure is used that allows to increase global removal time by a factor of 2 for random points [Dev09].

The displacement of a vertex v at a point p to a new location p', first checks whether the triangulation embedding remains planar or not after moving v to p'. If yes, it moves v to p' and simply performs a sequence of flips to restore the Delaunay property, which is $O(d)$ where d is the degree of the vertex after the displacement. Otherwise, the displacement is done by inserting a vertex at the new location, and removing the obsolete vertex. The complexity is $O(n)$ in the worst case, but only $O(1+\delta\sqrt{n})$ for evenly distributed vertices in the unit square, where δ is the Euclidean distance between the new and old locations.

After having performed a point location, the nearest neighbor of a point is found in time $O(n)$ in the worst case, but in time $O(1)$ for vertices distributed uniformly at random and any query point.

37.5.2 Example: a Delaunay Terrain

The following code creates a Delaunay triangulation with the usual Euclidean metric for the vertical projection of a terrain model. The points have elevation, that is they are 3D points, but the predicates used to build the Delaunay triangulation are computed using only the x and y coordinates of these points.

The class *Projection_traits_xy_3<K>* is part of the 2D and 3D Linear Geometric Kernel, and replaces the class *Triangulation_euclidean_traits_xy_3<K>* which is deprecated.

```
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/Projection_traits_xy_3.h>
#include <CGAL/Delaunay_triangulation_2.h>

#include <fstream>


typedef CGAL::Exact_predicates_inexact_constructions_kernel K;
typedef CGAL::Projection_traits_xy_3<K>  Gt;
typedef CGAL::Delaunay_triangulation_2<Gt> Delaunay;

typedef K::Point_3   Point;

int main()
{
  std::ifstream in("data/terrain.cin");
  std::istream_iterator<Point> begin(in);
  std::istream_iterator<Point> end;

  Delaunay dt;
  dt.insert(begin, end);
  std::cout << dt.number_of_vertices() << std::endl;
  return 0;
}
```

File: examples/Triangulation_2/terrain.cpp

37.5.3 Example: Voronoi Diagram

The following code computes the edges of Voronoi diagram of a set of data points and counts the number of finite edges and the number of rays of this diagram

```
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/Delaunay_triangulation_2.h>

#include <fstream>

typedef CGAL::Exact_predicates_inexact_constructions_kernel K;

typedef CGAL::Delaunay_triangulation_2<K>  Triangulation;
typedef Triangulation::Edge_iterator  Edge_iterator;
typedef Triangulation::Point          Point;

int main( )
{
  std::ifstream in("data/voronoi.cin");
  std::istream_iterator<Point> begin(in);
  std::istream_iterator<Point> end;
  Triangulation T;
  T.insert(begin, end);

  int ns = 0;
  int nr = 0;
  Edge_iterator eit =T.edges_begin();
  for ( ; eit !=T.edges_end(); ++eit) {
    CGAL::Object o = T.dual(eit);
    if (CGAL::object_cast<K::Segment_2>(&o)) {++ns;}
    else if (CGAL::object_cast<K::Ray_2>(&o)) {++nr;}
  }
  std::cout << "The Voronoi diagram has " << ns << " finite edges "
            << " and " << nr << " rays" << std::endl;
  return 0;
}
```

File: examples/Triangulation_2/voronoi.cpp

37.5.4 Example: Print Voronoi Diagram Edges Restricted to a rectangle

The following code computes the Delaunay triangulation of a set of points and prints the Voronoi edges restricted to a given rectangle.

```
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/Delaunay_triangulation_2.h>
#include <iterator>
```

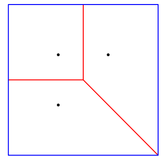


Figure 37.6: Voronoi diagram (in red) of the black points restricted to the blue rectangle.

```
typedef CGAL::Exact_predicates_inexact_constructions_kernel K;
typedef K::Point_2 Point_2;
typedef K::Iso_rectangle_2 Iso_rectangle_2;
typedef K::Segment_2 Segment_2;
typedef K::Ray_2 Ray_2;
typedef K::Line_2 Line_2;
typedef CGAL::Delaunay_triangulation_2<K>  Delaunay_triangulation_2;

//A class to recover Voronoi diagram from stream.
//Rays, lines and segments are cropped to a rectangle
//so that only segments are stored
struct Cropped_voronoi_from_delaunay{
  std::list<Segment_2> m_cropped_vd;
  Iso_rectangle_2 m_bbox;

  Cropped_voronoi_from_delaunay(const Iso_rectangle_2&
   bbox):m_bbox(bbox){}

  template <class RSL>
  void crop_and_extract_segment(const RSL& rsl){
    CGAL::Object obj = CGAL::intersection(rsl,m_bbox);
    const Segment_2* s=CGAL::object_cast<Segment_2>(&obj);
    if (s) m_cropped_vd.push_back(*s);
  }

  void operator<<(const Ray_2& ray)    { crop_and_extract_segment(ray); }
  void operator<<(const Line_2& line)  { crop_and_extract_segment(line);
   }
  void operator<<(const Segment_2& seg){ crop_and_extract_segment(seg); }
};

int main(){
  //consider some points
  std::vector<Point_2> points;
  points.push_back(Point_2(0,0));
  points.push_back(Point_2(1,1));
  points.push_back(Point_2(0,1));

  Delaunay_triangulation_2 dt2;
  //insert points into the triangulation
  dt2.insert(points.begin(),points.end());
  //construct a rectangle
  Iso_rectangle_2 bbox(-1,-1,2,2);
  Cropped_voronoi_from_delaunay vor(bbox);
  //extract the cropped Voronoi diagram
```

```
  dt2.draw_dual(vor);
  //print the cropped Voronoi diagram as segments
  std::copy(vor.m_cropped_vd.begin(),vor.m_cropped_vd.end(),
    std::ostream_iterator<Segment_2>(std::cout,"\n"));
}
```

File: examples/Triangulation_2/print_cropped_voronoi.cpp

37.6 Regular Triangulations

37.6.1 Description

Let $PW = \{(p_i, w_i) | i = 1, \ldots, n\}$ be a set of weighted points where each p_i is a point and each w_i is a scalar called the weight of point p_i. Alternatively, each weighted point (p_i, w_i) can be regarded as a sphere (or a circle, depending on the dimensionality of p_i) with center p_i and radius $r_i = \sqrt{w_i}$.

The power diagram of the set PW is a space partition in which each cell corresponds to a sphere (p_i, w_i) of PW and is the locus of points p whose power with respect to (p_i, w_i) is less than its power with respect to any other sphere in PW. In the two-dimensional space, the dual of this diagram is a triangulation whose domain covers the convex hull of the set $P = \{p_i | i = 1, \ldots, n\}$ of center points and whose vertices form a subset of P. Such a triangulation is called a regular triangulation. Three points p_i, p_j and p_k of P form a triangle in the regular triangulation of PW iff there is a point p of the plane with equal powers with respect to (p_i, w_i), (p_j, w_j) and (p_k, w_k) and such that this power is less than the power of p with respect to any other sphere in PW.

Let us defined the power product of two weighted points (p_i, w_i) and (p_j, w_j) as:

$$\Pi(p_i, w_i, p_j, w_j) = p_i p_j^2 - w_i - w_j.$$

$\Pi(p_i, w_i, p_j, 0)$ is simply the power of point p_j with respect to the sphere (p_i, w_i), and two weighted points are said to be orthogonal if their power product is null. The power circle of three weighted points (p_i, w_i), (p_j, w_j) and (p_k, w_k) is defined as the unique circle (π, ω) orthogonal to (p_i, w_i), (p_j, w_j) and (p_k, w_k).

The regular triangulation of the sets PW satisfies the following *regular property* (which just reduces to the Delaunay property when all the weights are null): a triangle $p_i p_j p_k$ is a face of the regular triangulation of PW iff the power product of any weighted point (p_l, w_l) of PW with the power circle of (p_i, w_i), (p_j, w_j) and (p_k, w_k) is positive or null. We call power test of (p_i, w_i), (p_j, w_j), (p_k, w_k), and (p_l, w_l), the predicates which amount to compute the sign of the power product of (p_l, w_l) with respect to the power circle of (p_i, w_i), (p_j, w_j) and (p_k, w_k). This predicate amounts to computing the sign of the following determinant

$$\begin{vmatrix} 1 & x_i & y_i & x_i^2 + y_i^2 - w_i \\ 1 & x_j & y_j & x_j^2 + y_j^2 - w_j \\ 1 & x_k & y_k & x_k^2 + y_k^2 - w_k \\ 1 & x_l & y_l & x_l^2 + y_l^2 - w_l \end{vmatrix}$$

A pair of neighboring faces $p_i p_j p_k$ and $p_i p_j p_l$ is said to be locally regular (with respect to the weights in PW) if the power test of (p_i, w_i), (p_j, w_j), (p_k, w_k), and (p_l, w_l) is positive. A classical result of computational geometry establishes that a triangulation of the convex hull of P such that any pair of neighboring faces is regular with respect to PW, is a regular triangulation of PW.

Alternatively, the regular triangulation of the weighted points set PW can be obtained as the projection on the two dimensional plane of the convex hull of the set of three dimensional points $P' = \{(p_i, p_i^2 - w_i) | i = 1, \ldots, n\}$.

The class *Regular_triangulation_2<Traits, Tds>* is designed to maintain the regular triangulation of a set of $2d$ weighted points. It derives from the class *Triangulation_2<Traits, Tds>*. The functions *insert* and *remove* are overwritten to handle weighted points and maintain the regular property. The function *move* is not overwritten and thus does not preserve the regular property. The vertices of the regular triangulation of a set of weighted points *PW* correspond only to a subset of *PW*. Some of the input weighted points have no cell in the dual power diagrams and therefore do not correspond to a vertex of the regular triangulation. Such a point is called a hidden point. Because hidden points can reappear later on as vertices when some other point is removed, they have to be stored somewhere. The regular triangulation store those points in special vertices, called hidden vertices. A hidden point can reappear as vertex of the triangulation only when the two dimensional face that hides it is removed from the triangulation. To deal with this feature, each face of a regular triangulation stores a list of hidden vertices. The points in those vertices are reinserted in the triangulation when the face is removed.

Regular triangulation have member functions to construct the vertices and edges of the dual power diagrams.

The Geometric Traits

The geometric traits of a regular triangulation must provide a weighted point type and a power test on these weighted points. The concept *RegularTriangulationTraits_2*, is a refinement of the concept *TriangulationTraits_2*. CGAL provides the class *Regular_triangulation_euclidean_traits_2<Rep,Weight>* which is a model for the traits concept *RegularTriangulationTraits_2*. The class *Regular_triangulation_euclidean_traits_2<Rep,Weight>* derives from the class *Triangulation_euclidean_traits_2<Rep>* and uses a *Weighted_point* type derived from the type *Point_2* of *Triangulation_euclidean_traits_2<Rep>*.

Note that, since the type *Weighted_point* is not defined in CGAL kernels, plugging a filtered kernel such as *Exact_predicates_exact_constructions_kernel* in *Regular_triangulation_euclidean_traits_2<K,Weight>* will in fact not provide exact predicates on weighted points. To solve this, there is also another model of the traits concept, *Regular_triangulation_filtered_traits_2<FK>*, which is providing filtered predicates (exact and efficient). The argument *FK* must be a model of the *Kernel* concept, and it is also restricted to be a instance of the *Filtered_kernel* template.

The Vertex Type and Face Type of a Regular Triangulation

The base vertex type of a regular triangulation includes a Boolean data member to mark the hidden state of the vertex. Therefore CGAL defines the concept *RegularTriangulationVertexBase_2* which refine the concept *TriangulationVertexBase_2* and provides a default model for this concept.

The base face type of a regular triangulation is required to provide a list of hidden vertices, designed to store the points hidden by the face. It has to be a model of the concept *RegularTriangulationFaceBase_2*. CGAL provides the templated class *Regular_triangulation_face_base_2<Traits>* as a default base class for faces of regular triangulations.

37.6.2 Example: a Regular Triangulation

The following code creates a regular triangulation of a set of weighted points and output the number of vertices and the number of hidden vertices.

```
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/Regular_triangulation_euclidean_traits_2.h>
#include <CGAL/Regular_triangulation_filtered_traits_2.h>
#include <CGAL/Regular_triangulation_2.h>
```

```
#include <fstream>

typedef CGAL::Exact_predicates_inexact_constructions_kernel K;
typedef CGAL::Regular_triangulation_filtered_traits_2<K>  Traits;
typedef CGAL::Regular_triangulation_2<Traits> Regular_triangulation;

int main()
{
   Regular_triangulation rt;
   std::ifstream in("data/regular.cin");

   Regular_triangulation::Weighted_point wp;
   int count = 0;
   while(in >> wp){
       count++;
     rt.insert(wp);
   }
   rt.is_valid();
   std::cout << "number of inserted points : " << count << std::endl;
   std::cout << "number of vertices :  " ;
   std::cout << rt.number_of_vertices() << std::endl;
   std::cout << "number of hidden vertices :  " ;
   std::cout << rt.number_of_hidden_vertices() << std::endl;
   return 0;
}
```

File: examples/Triangulation_2/regular.cpp

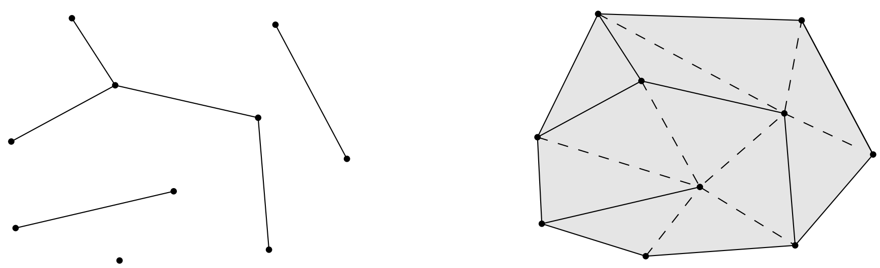
37.7 Constrained Triangulations

A constrained triangulation is a triangulation of a set of points that has to include among its edges a given set of segments joining the points. The corresponding edges are called *constrained edges*.

The endpoints of constrained edges are of course vertices of the triangulation. However the triangulation may include include other vertices as well. There are three versions of constrained triangulations.

- In the basic version, the constrained triangulation does not handle intersecting constraints, and the set of input constraints is required to be a set of segments that do not intersect except possibly at their endpoints. Any number of constrained edges are allowed to share the same endpoint. Vertical constrained edges or constrained edges with null length are allowed.
- The two other versions support intersecting input constraints. In those versions, input constraints are allowed to be intersecting, overlapping or partially overlapping segments. The triangulation introduces additional vertices at each point that is the proper intersection points of two constraints. A single constraint intersecting other constraints will then appear as several edges in the triangulation. The two versions dealing with intersecting constraints differ in the way intersecting constraints are dealt with.
  - One of them is designed to be robust when predicates are evaluated exactly but constructions (i. e. intersection computations) are approximate.

– The other one is designed to be used with exact arithmetic (meaning exact evaluation of predicates and exact computation of intersections.) This last version finds its full efficiency when used in conjunction with a constraint hierarchy data structure (which allows one to avoid the cascading of intersection computations) as provided in the class *Constrained_triangulation_plus_2*. See section 37.9.



A constrained triangulation is represented in the CGAL library as an object of the class *Constrained_triangulation_2<Traits,Tds,Itag>*. The third parameter *Itag* is the intersection tag which serves to choose how intersecting constraints are dealt with. This parameter has to be instantiated by one of the following classes :
CGAL::No_intersection_tag when input constraints do not intersect
CGAL::Exact_predicates_tag if the geometric traits provides exact predicates but approximate constructions
CGAL::Exact_intersections_tag when an exact predicates and exact constructions are provided.

The class *Constrained_triangulation_2<Traits,Tds, Itag>* inherits from *Triangulation_2<Traits,Tds>*. It defines an additional type *Constraint* to represent the constraints. A constraint is represented as a pair of points.

A constrained triangulation can be created from a list of constrained edges. The class *Constrained_triangulation_2<Traits,Tds,Itag>* overrides the insertion and removal of a point to take care of the information about constrained edges. The class also allows inline insertion of a new constraint, given by its two endpoints or the removal of a constraint. In current version, function *move* is not overwritten and thus does not take care of the constraints.

The Geometric Traits

The geometric traits of a constraint triangulation has to be a model of the concept *TriangulationTraits_2*. When intersections of input constraints are supported, the geometric traits class has to be a model of the concept *ConstrainedTriangulationTraits_2* which refines the concept *TriangulationTraits_2* providing additional function object types to compute the intersection of two segments.

The Base Face of a Constrained Triangulation

The information about constrained edges is stored in the faces of the triangulation. The base face of a Constrained Triangulation has to be a model for the concept *ConstrainedTriangulationFaceBase_2* which refines the concept *TriangulationFaceBase_2*. The concept *ConstrainedTriangulationFaceBase_2* requires member functions the get and set the constrained status of the edges.

CGAL provides a default base face class for constrained triangulations. This class, named *Constrained_triangulation_face_base_2<Traits>*, derives from the class *Triangulation_face_base_2<Traits>* and adds three Boolean data members to store the status of its edges.

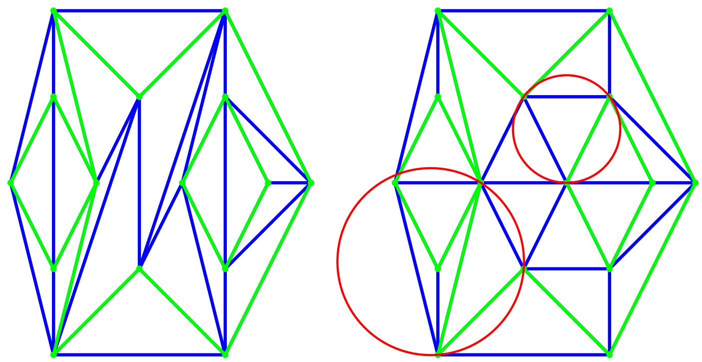


Figure 37.7: Constrained and Constrained Delaunay triangulation: the constraining edges are the green edges, a constrained triangulation is shown on the left, the constrained Delaunay triangulation with two examples of circumcircles is shown on the right.

37.8 Constrained Delaunay Triangulations

A constrained Delaunay triangulation is a triangulation with constrained edges which tries to be as much Delaunay as possible. As constrained edges are not necessarily Delaunay edges, the triangles of a constrained Delaunay triangulation do not necessarily fulfill the empty circle property but they fulfill a weaker *constrained empty circle property*. To state this property, it is convenient to think of constrained edges as blocking the view. Then, a triangulation is constrained Delaunay iff the circumscribing circle of any facet encloses no vertex visible from the interior of the facet. As in the case of constrained triangulations, three different versions of Delaunay constrained triangulations are provided. The first version handle set of constraints which do not intersect except possibly at the endpoints. The two other versions handle intersecting input constraints. One of them is designed to be designed to be robust when used in conjunction with a geometric traits providing exact predicates and approximate constructions (such as a *CGAL::Filtered_Kernel* or any kernel providing filtered exact predicates). The third version is designed to be used with an exact arithmetic number type.

The CGAL class *Constrained_Delaunay_triangulation_2<Traits,Tds,Itag>* is designed to represent constrained Delaunay triangulations.

As in the case of constraints triangulation, the third parameter *Itag* is the intersection tag and serves to choose how intersecting constraints are dealt with. It can be instantiated with one of the following class : *CGAL::No_intersection_tag*, *CGAL::Exact_predicates_tag*, *CGAL::Exact_intersections_tag* (see Section 37.7.

A constrained Delaunay triangulation is not a Delaunay triangulation but it is a constrained triangulation. Therefore the class *Constrained_Delaunay_triangulation_2<Traits,Tds,Itag>* derives from the class *Constrained_triangulation_2<Traits,Tds,Itag>*.

The constrained Delaunay triangulation has member functions to override the insertion and removal of a point or of a constraint. Each of those member function takes care to restore the constrained empty circle property.

The Geometric Traits

The geometric traits of a constrained Delaunay triangulation is required to provide the *side_of_oriented_circle* predicate as the geometric traits of a Delaunay triangulation and has to a model of the concept *DelaunayTriangulationTraits_2*. When intersecting input constraints are supported the geometric traits is further required to provide function objects to compute constraints intersections. Then the geometric traits has to be at the same time a model of the concept *ConstrainedTriangulationTraits_2*.

The Face Base Class

Information about the status (constrained or not) of the edges of the triangulation has to be stored in the face class and the base face class of a constrained Delaunay triangulation has to be a model of *ConstrainedTriangulationFaceBase_2*.

37.8.1 Example: a Constrained Delaunay Triangulation

The following code inserts a set of intersecting constraint segments into a triangulation and counts the number of constrained edges of the resulting triangulation.

```
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/Constrained_Delaunay_triangulation_2.h>

#include <cassert>
#include <iostream>

typedef CGAL::Exact_predicates_inexact_constructions_kernel K;

typedef CGAL::Triangulation_vertex_base_2<K>                     Vb;
typedef CGAL::Constrained_triangulation_face_base_2<K>           Fb;
typedef CGAL::Triangulation_data_structure_2<Vb,Fb>              TDS;
typedef CGAL::Exact_predicates_tag                               Itag;
typedef CGAL::Constrained_Delaunay_triangulation_2<K, TDS, Itag> CDT;
typedef CDT::Point          Point;

int
main( )
{
  CDT cdt;
  std::cout << "Inserting a grid of 5x5 constraints " << std::endl;
  for (int i = 1; i < 6; ++i)
    cdt.insert_constraint( Point(0,i), Point(6,i));
  for (int j = 1; j < 6; ++j)
    cdt.insert_constraint( Point(j,0), Point(j,6));

  assert(cdt.is_valid());
  int count = 0;
  for (CDT::Finite_edges_iterator eit = cdt.finite_edges_begin();
       eit != cdt.finite_edges_end();
       ++eit)
    if (cdt.is_constrained(*eit)) ++count;
  std::cout << "The number of resulting constrained edges is  ";
  std::cout <<  count << std::endl;
  return 0;
}
```

File: examples/Triangulation_2/constrained.cpp

37.8.2 Example: Triangulating a Polygonal Domain

The following code inserts two nested polygons into a constrained Delaunay triangulation and counts the number of facets that are inside the domain delimited by these polygons. Note that the following code does not work if the boundaries of the polygons intersect.

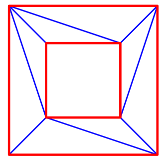


Figure 37.8: Triangulation (in blue) of the domain delimited by the red polygons.

```
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/Constrained_Delaunay_triangulation_2.h>
#include <CGAL/Triangulation_face_base_with_info_2.h>
#include <CGAL/Polygon_2.h>
#include <iostream>

struct FaceInfo2
{
  FaceInfo2(){}
  int nesting_level;

  bool in_domain(){
    return nesting_level%2 == 1;
  }
};


typedef CGAL::Exact_predicates_inexact_constructions_kernel       K;
typedef CGAL::Triangulation_vertex_base_2<K>                      Vb;
typedef CGAL::Triangulation_face_base_with_info_2<FaceInfo2,K>    Fbb;
typedef CGAL::Constrained_triangulation_face_base_2<K,Fbb>        Fb;
typedef CGAL::Triangulation_data_structure_2<Vb,Fb>               TDS;
typedef CGAL::Exact_predicates_tag                                Itag;
typedef CGAL::Constrained_Delaunay_triangulation_2<K, TDS, Itag>  CDT;
typedef CDT::Point                                                Point;
typedef CGAL::Polygon_2<K>
   Polygon_2;

void
mark_domains(CDT& ct,
             CDT::Face_handle start,
             int index,
             std::list<CDT::Edge>& border )
{
  if(start->info().nesting_level != -1){
    return;
  }
  std::list<CDT::Face_handle> queue;
  queue.push_back(start);
```

```
  while(! queue.empty()){
    CDT::Face_handle fh = queue.front();
    queue.pop_front();
    if(fh->info().nesting_level == -1){
      fh->info().nesting_level = index;
      for(int i = 0; i < 3; i++){
        CDT::Edge e(fh,i);
        CDT::Face_handle n = fh->neighbor(i);
        if(n->info().nesting_level == -1){
          if(ct.is_constrained(e)) border.push_back(e);
          else queue.push_back(n);
        }
      }
    }
  }
}

//explore set of facets connected with non constrained edges,
//and attribute to each such set a nesting level.
//We start from facets incident to the infinite vertex, with a nesting
//level of 0. Then we recursively consider the non-explored facets
    incident
//to constrained edges bounding the former set and increase the nesting
    level by 1.
//Facets in the domain are those with an odd nesting level.
void
mark_domains(CDT& cdt)
{
  for(CDT::All_faces_iterator it = cdt.all_faces_begin(); it !=
   cdt.all_faces_end(); ++it){
    it->info().nesting_level = -1;
  }

  int index = 0;
  std::list<CDT::Edge> border;
  mark_domains(cdt, cdt.infinite_face(), index++, border);
  while(! border.empty()){
    CDT::Edge e = border.front();
    border.pop_front();
    CDT::Face_handle n = e.first->neighbor(e.second);
    if(n->info().nesting_level == -1){
      mark_domains(cdt, n, e.first->info().nesting_level+1, border);
    }
  }
}

void insert_polygon(CDT& cdt,const Polygon_2& polygon){
  if ( polygon.is_empty() ) return;
  CDT::Vertex_handle
   v_prev=cdt.insert(*CGAL::cpp0x::prev(polygon.vertices_end()));
  for (Polygon_2::Vertex_iterator vit=polygon.vertices_begin();
       vit!=polygon.vertices_end();++vit)
  {
    CDT::Vertex_handle vh=cdt.insert(*vit);
```

```
    cdt.insert_constraint(vh,v_prev);
    v_prev=vh;
  }
}

int main( )
{
  //construct two non-intersecting nested polygons
  Polygon_2 polygon1;
  polygon1.push_back(Point(0,0));
  polygon1.push_back(Point(2,0));
  polygon1.push_back(Point(2,2));
  polygon1.push_back(Point(0,2));
  Polygon_2 polygon2;
  polygon2.push_back(Point(0.5,0.5));
  polygon2.push_back(Point(1.5,0.5));
  polygon2.push_back(Point(1.5,1.5));
  polygon2.push_back(Point(0.5,1.5));

  //Insert the polyons into a constrained triangulation
  CDT cdt;
  insert_polygon(cdt,polygon1);
  insert_polygon(cdt,polygon2);

  //Mark facets that are inside the domain bounded by the polygon
  mark_domains(cdt);

  int count=0;
  for (CDT::Finite_faces_iterator fit=cdt.finite_faces_begin();
                                  fit!=cdt.finite_faces_end();++fit)
  {
    if ( fit->info().in_domain() ) ++count;
  }

  std::cout << "There are " << count << " facets in the domain." <<
   std::endl;

  return 0;
}
```

File: examples/Triangulation_2/polygon_triangulation.cpp

37.9 Constrained Triangulations Plus

The class *Constrained_triangulation_plus_2<Tr>* provides a constrained triangulation with an additional data structure called the constraint hierarchy that keeps track of the input constraints and of their refinement in the triangulation. The class *Constrained_triangulation_plus_2<Tr>* inherits from its template parameter Tr, which has to be instantiated by a constrained or constrained Delaunay triangulation. According to its intersection tag, the base class will support intersecting input constraints or not. When intersections of input constraints are supported, the base class constructs a triangulation of the arrangement of the constraints, introducing new vertices at each proper intersection points and refining the input constraints into sub-constraints which appear as edges (more precisely as constrained edges) of the triangulation. The data structure maintains for each input constraint the sequence of intersection vertices added on this constraint. The constraint hierarchy also allows the user to retrieve the set of constrained edges of the triangulation, and for each constrained edge, the set of input constraints that overlap it.

The class *Constrained_triangulation_plus_2<Tr>* is especially useful when the base constrained triangulation class handles intersections of constraints and uses an exact number type, i.e. when its intersection tag is *CGAL::Exact_intersections_tag*. Indeed in this case, the *Constrained_triangulation_plus_2<Tr>* is specially designed to avoid cascading in the computations of intersection points.

37.9.1 Example: Building a Triangulated Arrangement of Segments

The following code inserts a set of intersecting constraint segments into a triangulation and counts the number of constrained edges of the resulting triangulation.

```
#include <CGAL/Exact_predicates_exact_constructions_kernel.h>
#include <CGAL/intersections.h>
#include <CGAL/Constrained_Delaunay_triangulation_2.h>
#include <CGAL/Constrained_triangulation_plus_2.h>

#include <cassert>
#include <iostream>

typedef CGAL::Exact_predicates_exact_constructions_kernel K;

typedef CGAL::Triangulation_vertex_base_2<K>              Vb;
typedef CGAL::Constrained_triangulation_face_base_2<K>    Fb;
typedef CGAL::Triangulation_data_structure_2<Vb,Fb>       TDS;
typedef CGAL::Exact_intersections_tag                     Itag;
typedef CGAL::Constrained_Delaunay_triangulation_2<K,TDS,Itag> CDT;
typedef CGAL::Constrained_triangulation_plus_2<CDT>       CDTplus;
typedef CDTplus::Point                                    Point;

int
main( )
{
  CDTplus cdt;
  std::cout  << "Inserting a grid 5 x 5 of constraints " << std::endl;
  for (int i = 1; i < 6; ++i)
    cdt.insert_constraint( Point(0,i), Point(6,i));
  for (int j = 1; j < 6; ++j)
    cdt.insert_constraint( Point(j,0), Point(j,6));

  assert(cdt.is_valid());
  int count = 0;
  for (CDTplus::Subconstraint_iterator scit = cdt.subconstraints_begin();
       scit != cdt.subconstraints_end();
       ++scit)  ++count;
  std::cout << "The number of resulting constrained edges is  "
            <<  count << std::endl;
  return 0;
}
```

File: examples/Triangulation_2/constrained_plus.cpp

37.10 The Triangulation Hierarchy

The class *Triangulation_hierarchy_2<Tr>* implements a triangulation augmented with a data structure to answer efficiently point location queries. The data structure is a hierarchy of triangulations. The triangulation at the lowest level is the original triangulation where operations and point location are to be performed. Then at each succeeding level, the data structure stores a triangulation of a small random sample of the vertices of the triangulation at the preceding level. Point location is done through a top down nearest neighbor query. The nearest neighbor query is first performed naively in the top level triangulation. Then, at each following level, the nearest neighbor at that level is found through a linear walk performed from the nearest neighbor found at the preceding level. Because the number of vertices in each triangulation is only a small fraction of the number of vertices of the preceding triangulation, the data structure remains small and achieves fast point location queries on real data. As proved in [Dev98], this structure has an optimal behavior when it is built for Delaunay triangulations. However it can be used as well for other triangulations and the class *Triangulation_hierarchy_2<Tr>* is templated by a parameter which is to be instantiated by one of the CGAL triangulation classes.

The class *Triangulation_hierarchy_2<Tr>* inherits from the triangulation type passed as template parameter *Tr*. The *insert*, *move*, and *remove* member functions are overwritten to update the data structure at each operation. The locate queries are also overwritten to take advantage of the data structure for a fast processing.

The Vertex of a Triangulation Hierarchy

The base vertex class of a triangulation hierarchy has to be a model of the concept *TriangulationHierarchyVertexBase_2* which extends the concept *TriangulationVertexBase_2*. This extension adds access and setting member functions for two pointers to the corresponding vertices in the triangulations of the next and preceding levels.

CGAL provides the class *Triangulation_hierarchy_vertex_base_2<Vb>* which is a model for the concept *TriangulationHierarchyVertexBase_2*. This class is templated by a parameter *Vb* which is to be instantiated by a model of the concept *TriangulationVertexBase_2*. The class *Triangulation_hierarchy_vertex_base_2<Vb>* inherits from its template parameter *Vb*. This design allows to use for *Vb* either the default vertex class or a user customized vertex class with additional functionalities.

37.10.1 Examples For the Use of a Triangulation Hierarchy

The following program is example for the standard use of a triangulation hierarchy to enhance the efficiency of a Delaunay triangulation. The program outputs the number of vertices at the different levels of the hierarchy.

```
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/Delaunay_triangulation_2.h>
#include <CGAL/Triangulation_hierarchy_2.h>
#include <CGAL/point_generators_2.h>
#include <CGAL/algorithm.h>
#include <cassert>

typedef CGAL::Exact_predicates_inexact_constructions_kernel K;

typedef CGAL::Triangulation_vertex_base_2<K>              Vbb;
```

```
typedef CGAL::Triangulation_hierarchy_vertex_base_2<Vbb> Vb;
typedef CGAL::Triangulation_face_base_2<K>               Fb;
typedef CGAL::Triangulation_data_structure_2<Vb,Fb>      Tds;
typedef CGAL::Delaunay_triangulation_2<K,Tds>            Dt;
typedef CGAL::Triangulation_hierarchy_2<Dt>              Triangulation;
typedef Triangulation::Point                             Point;
typedef CGAL::Creator_uniform_2<double,Point>            Creator;

int main( )
{
  std::cout << "insertion of 1000 random points" << std::endl;
  Triangulation t;
  CGAL::Random_points_in_square_2<Point,Creator> g(1.);
  CGAL::cpp0x::copy_n( g, 1000, std::back_inserter(t));

  //verbose mode of is_valid ; shows the number of vertices at each
   level
  std::cout << "The number of vertices at successive levels" <<
   std::endl;
  assert(t.is_valid(true));

  return 0;
}
```

File: examples/Triangulation_2/hierarchy.cpp

The following program shows how to use a triangulation hierarchy in conjunction with a constrained triangulation plus.

```
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/Constrained_Delaunay_triangulation_2.h>
#include <CGAL/Triangulation_hierarchy_2.h>
#include <CGAL/Constrained_triangulation_plus_2.h>

#include <cassert>
#include <iostream>

typedef CGAL::Exact_predicates_inexact_constructions_kernel K;

typedef CGAL::Triangulation_vertex_base_2<K>             Vbb;
typedef CGAL::Triangulation_hierarchy_vertex_base_2<Vbb> Vb;
typedef CGAL::Constrained_triangulation_face_base_2<K>   Fb;
typedef CGAL::Triangulation_data_structure_2<Vb,Fb>      TDS;
typedef CGAL::Exact_predicates_tag                       Itag;
typedef CGAL::Constrained_Delaunay_triangulation_2<K,TDS,Itag> CDT;
typedef CGAL::Triangulation_hierarchy_2<CDT>             CDTH;
typedef CGAL::Constrained_triangulation_plus_2<CDTH>     Triangulation;

typedef Triangulation::Point                             Point;

int
main( )
{
```

```
  Triangulation cdt;
  std::cout << "Inserting a grid 5 x 5 of  constraints " << std::endl;
    for (int i = 1; i < 6; ++i)
    cdt.insert_constraint( Point(0,i), Point(6,i));
    for (int j = 1; j < 6; ++j)
    cdt.insert_constraint( Point(j,0), Point(j,6));

  int count = 0;
  for (Triangulation::Subconstraint_iterator scit =
    cdt.subconstraints_begin();
        scit != cdt.subconstraints_end();
        ++scit)  ++count;
  std::cout << "The number of resulting constrained edges is  ";
  std::cout <<  count << std::endl;

  //verbose mode of is_valid ; shows the number of vertices at each
    level
  std::cout << "The number of vertices at successive levels" <<
    std::endl;
  assert(cdt.is_valid(true));

  return 0;
}
```

File: examples/Triangulation_2/constrained_hierarchy_plus.cpp

37.11 Flexibility

37.11.1 Using Customized Vertices and Faces

To be able to adapt to various needs, a highly flexible design has been selected for 2D triangulations. We have already seen that the triangulation classes have two parameters: a geometric traits class and a triangulation data structure class which the user can instantiate with his own customized classes.

The most useful flexibility however comes from the fact that the triangulation data structure itself has two template parameters to be instantiated by classes for the vertices and faces of the triangulation. Using his own customized classes to instantiate these parameters, the user can easily build up a triangulation with additional information or functionality in the vertices and faces.

A Cyclic Dependency

To insure flexibility, the triangulation data structure is templated by the vertex and face base classes. Also since incidence and adjacency relations are stored in vertices and faces, the base classes have to know the types of handles on vertices and faces provided by the triangulation data structure. Thus the vertex and base classes have to be themselves parameterized by the triangulation data structure, and there is a cyclic dependency on template parameter.

Previously, this cyclic dependency was avoided by using only *void** pointers in the interface of base classes. These *void** were converted to appropriate types at the triangulation data structure levels. This solution had

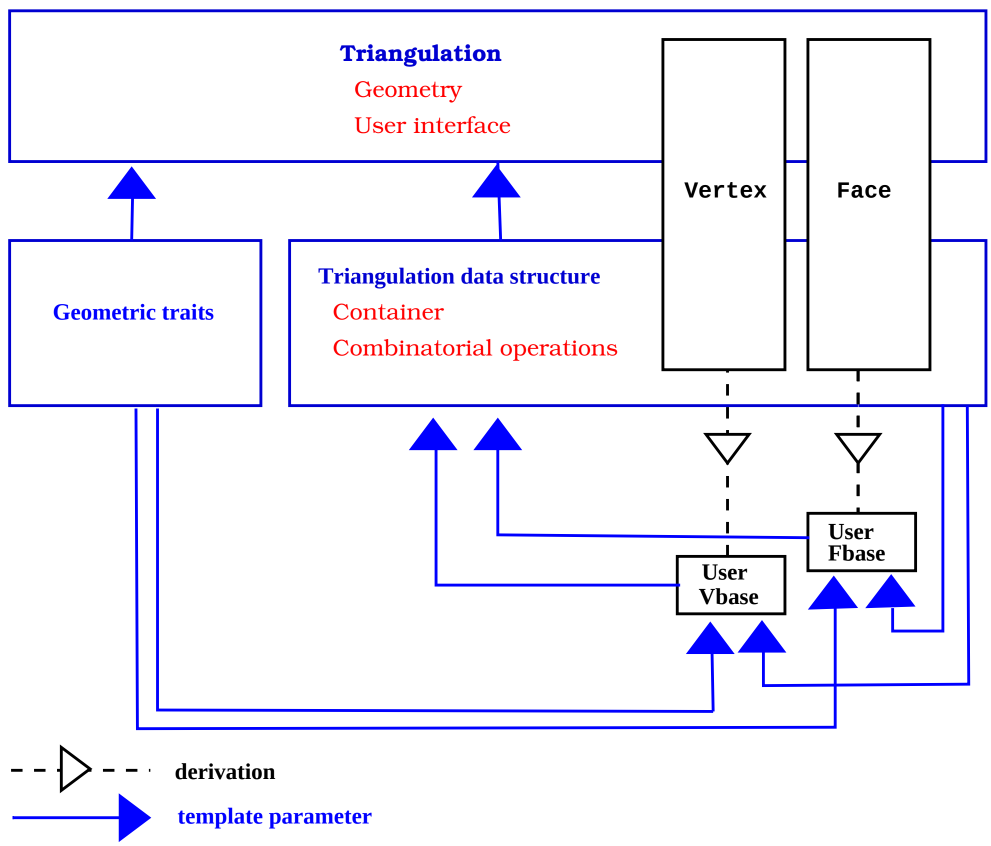


Figure 37.9: The cyclic dependency in triangulations software design.

some drawbacks : mainly the user could not add in the vertices or faces of the triangulation a functionality related to types defined by the triangulation data structure, for instance a handle to a vertex, and he was lead to use himself *void** pointers). The new solution to resolve the template dependency is based on a rebind mechanism similar to the mechanism used in the standard allocator class std::allocator. The rebind mechanism is described in Section 38.3 of Chapter 38. For now, we will just notice that the design requires the existence in the vertex and face base classes of a nested template class *Rebind_TDS* defining a type *Other* used by the rebinding mechanism.

The two following examples show how the user can put in use the flexibility offered by the base classes parameters.

Adding Colors

The first example corresponds to a case where the user wishes to add in the vertices or faces of the triangulation an additional information that does not depend on types provided by the triangulation data structure. In that case, predefined classes *Triangulation_vertex_base_with_info_2<Info,Traits,Vb>* or *Triangulation_face_base_with_info_2<Info,Traits,Vb>* can be used. Those classes have a template parameter *Info* devoted to handle additional information. The following examples shows how to add a *CGAL::Color* in the triangulation faces.

```
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/IO/Color.h>
#include <CGAL/Triangulation_2.h>
#include <CGAL/Triangulation_face_base_with_info_2.h>

typedef CGAL::Exact_predicates_inexact_constructions_kernel K;

typedef CGAL::Triangulation_vertex_base_2<K> Vb;
typedef CGAL::Triangulation_face_base_with_info_2<CGAL::Color,K> Fb;
typedef CGAL::Triangulation_data_structure_2<Vb,Fb> Tds;
typedef CGAL::Triangulation_2<K,Tds> Triangulation;

typedef Triangulation::Face_handle Face_handle;
typedef Triangulation::Finite_faces_iterator Finite_faces_iterator;
typedef Triangulation::Point  Point;

int main() {
  Triangulation t;
  t.insert(Point(0,1));
  t.insert(Point(0,0));
  t.insert(Point(2,0));
  t.insert(Point(2,2));

  Finite_faces_iterator fc = t.finite_faces_begin();
  for( ; fc != t.finite_faces_end(); ++fc)  fc->info() = CGAL::BLUE;

  Point p(0.5,0.5);
  Face_handle fh = t.locate(p);
  fh->info() = CGAL::RED;

  return 0;
}
```

File: examples/Triangulation_2/colored_face.cpp

Adding Handles

The second example shows how the user can still derive and plug in his own vertex or face class when he would like to have additional functionalities depending on types provided by the triangulation data structure.

```
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/Triangulation_2.h>
#include <cassert>

/* A vertex class with an additionnal handle */
template < class Gt, class Vb = CGAL::Triangulation_vertex_base_2<Gt> >
class My_vertex_base
  : public  Vb
{
  typedef Vb                                   Base;
public:
  typedef typename Vb::Vertex_handle           Vertex_handle;
  typedef typename Vb::Face_handle             Face_handle;
  typedef typename Vb::Point                   Point;

  template < typename TDS2 >
  struct Rebind_TDS {
    typedef typename Vb::template Rebind_TDS<TDS2>::Other    Vb2;
    typedef My_vertex_base<Gt,Vb2>                           Other;
  };

private:
  Vertex_handle  va_;

public:
  My_vertex_base() : Base() {}
  My_vertex_base(const Point & p) : Base(p) {}
  My_vertex_base(const Point & p, Face_handle f) : Base(f,p) {}
  My_vertex_base(Face_handle f) : Base(f) {}

  void set_associated_vertex(Vertex_handle va) { va_ = va;}
  Vertex_handle get_associated_vertex() {return va_ ; }
};

typedef CGAL::Exact_predicates_inexact_constructions_kernel K;

typedef My_vertex_base<K> Vb;
typedef CGAL::Triangulation_data_structure_2<Vb> Tds;
typedef CGAL::Triangulation_2<K,Tds> Triangulation;

typedef Triangulation::Vertex_handle Vertex_handle;
typedef Triangulation::Finite_faces_iterator Finite_faces_iterator;
typedef Triangulation::Point   Point;

int main() {
  Triangulation t;
  Vertex_handle v0 = t.insert(Point(0,1));
  Vertex_handle v1 = t.insert(Point(0,0));
  Vertex_handle v2 = t.insert(Point(2,0));
```

```
  Vertex_handle v3 = t.insert(Point(2,2));

  // associate vertices as you like
  v0->set_associated_vertex(v1);
  v1->set_associated_vertex(v2);
  v2->set_associated_vertex(v3);
  v3->set_associated_vertex(v0);
  assert( v0->get_associated_vertex() == v1);

  return 0;
}
```

File: examples/Triangulation_2/adding_handles.cpp

37.11.2 Setting Information While Inserting a Range of Points

The most efficient method to insert (weighted) points in a Delaunay (or regular) triangulation is to provide an iterator range over (weighted) points to the insert function. However, an iterator range of (weighted) points does not allow to set different information to each vertex. To solve this problem, in the case the vertex type of the triangulation is a model of the concept *TriangulationVertexBaseWithInfo_2* (such as *Triangulation_vertex_base_with_info_2*), we provide three examples doing the same operation: set an unsigned integer as the information of each vertex. The value of this unsigned integer is the initial order of the corresponding point given in the range.

Using an Iterator Over Pairs

Each point and its information are gathered into a pair. We provide the *insert* function of the triangulation with a range of such pairs.

```
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/Delaunay_triangulation_2.h>
#include <CGAL/Triangulation_vertex_base_with_info_2.h>
#include <vector>

typedef CGAL::Exact_predicates_inexact_constructions_kernel         K;
typedef CGAL::Triangulation_vertex_base_with_info_2<unsigned, K>    Vb;
typedef CGAL::Triangulation_data_structure_2<Vb>                    Tds;
typedef CGAL::Delaunay_triangulation_2<K, Tds>
   Delaunay;
typedef Delaunay::Point
   Point;

int main()
{
  std::vector< std::pair<Point,unsigned> > points;
  points.push_back( std::make_pair(Point(0,0),0)   );
  points.push_back( std::make_pair(Point(1,0),1)   );
  points.push_back( std::make_pair(Point(0,1),2)   );
  points.push_back( std::make_pair(Point(14,4),3)  );
  points.push_back( std::make_pair(Point(2,2),4)   );
  points.push_back( std::make_pair(Point(-4,0),5)  );
```

```
  Delaunay T;
  T.insert( points.begin(),points.end() );

  CGAL_assertion( T.number_of_vertices() == 6 );

  // check that the info was correctly set.
  Delaunay::Finite_vertices_iterator vit;
  for (vit = T.finite_vertices_begin(); vit != T.finite_vertices_end();
   ++vit)
    if( points[ vit->info() ].first != vit->point() ){
      std::cerr << "Error different info" << std::endl;
      exit(EXIT_FAILURE);
    }
  std::cout << "OK" << std::endl;

  return 0;
}
```

File: examples/Triangulation_2/info_insert_with_pair_iterator_2.cpp

Using the Boost Zip Iterator

Information and points are in separate containers. We use *boost::zip_iterator* to provide an iterator gathering them.

```
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/Delaunay_triangulation_2.h>
#include <CGAL/Triangulation_vertex_base_with_info_2.h>
#include <boost/iterator/zip_iterator.hpp>
#include <vector>

typedef CGAL::Exact_predicates_inexact_constructions_kernel         K;
typedef CGAL::Triangulation_vertex_base_with_info_2<unsigned, K>    Vb;
typedef CGAL::Triangulation_data_structure_2<Vb>                    Tds;
typedef CGAL::Delaunay_triangulation_2<K, Tds>
   Delaunay;
typedef Delaunay::Point
   Point;

int main()
{

  std::vector<unsigned> indices;
  indices.push_back(0);
  indices.push_back(1);
  indices.push_back(2);
  indices.push_back(3);
  indices.push_back(4);
  indices.push_back(5);
```

```
  std::vector<Point> points;
  points.push_back(Point(0,0));
  points.push_back(Point(1,0));
  points.push_back(Point(0,1));
  points.push_back(Point(1,47));
  points.push_back(Point(2,2));
  points.push_back(Point(-1,0));



  Delaunay T;
  T.insert( boost::make_zip_iterator(boost::make_tuple(
   points.begin(),indices.begin() )),
             boost::make_zip_iterator(boost::make_tuple(
   points.end(),indices.end() ) )  );

  CGAL_assertion( T.number_of_vertices() == 6 );


  // check that the info was correctly set.
  Delaunay::Finite_vertices_iterator vit;
  for (vit = T.finite_vertices_begin(); vit != T.finite_vertices_end();
   ++vit)
    if( points[ vit->info() ] != vit->point() ){
      std::cerr << "Error different info" << std::endl;
      exit(EXIT_FAILURE);
    }

  return 0;
}
```

File: examples/Triangulation_2/info_insert_with_zip_iterator_2.cpp

Using the Boost Transform Iterator

We define a functor *Auto_count* used together with *boost::transform_iterator* to set the order of each point in the range. Note that this is correct because the iterator is dereferenced only once per point during the insertion.

```
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/Delaunay_triangulation_2.h>
#include <CGAL/Triangulation_vertex_base_with_info_2.h>
#include <boost/iterator/transform_iterator.hpp>
#include <vector>

typedef CGAL::Exact_predicates_inexact_constructions_kernel         K;
typedef CGAL::Triangulation_vertex_base_with_info_2<unsigned, K>    Vb;
typedef CGAL::Triangulation_data_structure_2<Vb>                    Tds;
typedef CGAL::Delaunay_triangulation_2<K, Tds>
   Delaunay;
typedef Delaunay::Point
   Point;
```

```
//a functor that returns a std::pair<Point,unsigned>.
//the unsigned integer is incremented at each call to
//operator()
struct Auto_count : public std::unary_function<const
    Point&,std::pair<Point,unsigned> >{
  mutable unsigned i;
  Auto_count() : i(0){}
  std::pair<Point,unsigned> operator()(const Point& p) const {
    return std::make_pair(p,i++);
  }
};

int main()
{
  std::vector<Point> points;
  points.push_back(Point(0,0));
  points.push_back(Point(1,0));
  points.push_back(Point(0,1));
  points.push_back(Point(4,10));
  points.push_back(Point(2,2));
  points.push_back(Point(-1,0));


  Delaunay T;
  T.insert( boost::make_transform_iterator(points.begin(),Auto_count()),
            boost::make_transform_iterator(points.end(),  Auto_count() )
    );

  CGAL_assertion( T.number_of_vertices() == 6 );

  // check that the info was correctly set.
  Delaunay::Finite_vertices_iterator vit;
  for (vit = T.finite_vertices_begin(); vit != T.finite_vertices_end();
    ++vit)
    if( points[ vit->info() ] != vit->point() ){
      std::cerr << "Error different info" << std::endl;
      exit(EXIT_FAILURE);
    }
  std::cout << "OK" << std::endl;

  return 0;
}
```

File: examples/Triangulation_2/info_insert_with_transform_iterator_2.cpp

37.12 Design and Implementation History

The code of this package is the result of a long development process. Here follows a tentative list of people who added their stone to this package : Jean-Daniel Boissonnat, Hervé Brönnimann, Olivier Devillers, Andreas Fabri, Frédéric Fichel, Julia Flötotto, Monique Teillaud and Mariette Yvinec.

2D Triangulations Reference Manual

Mariette Yvinec

A triangulation is a 2-dimensional simplicial complex which is pure connected and without singularities. Thus a triangulation can be viewed as a collection of triangular faces, such that two faces either have an empty intersection or share an edge or a vertex.

The basic elements of the representation are vertices and faces. Each triangular face gives access to its three incident vertices and to its three adjacent faces. Each vertex gives access to one of its incident faces and through that face to the circular list of its incident faces. The edges are not explicitly represented, they are only represented through the adjacency relations of two faces.

The triangulation classes of CGAL depend on two template parameters. The first template parameter stands for a geometric traits class which is assumed to provide the geometric objects (points, segments and triangles) forming the triangulation and the geometric predicates on those objects. The second template parameter stands for a model of triangulation data structure acting as a container for faces and vertices while taking care of the combinatorial aspects of the triangulation. The concepts and models relative to the triangulation data structure are described in Chapter 38.3.4.

37.13 Classified Reference Pages

Concepts

Classes

Enum

37.14 Alphabetical List of Reference Pages

Concept

ConstrainedDelaunayTriangulationTraits_2

Definition

The concept ConstrainedDelaunayTriangulationTraits_2 defines the requirements for the geometric traits class of a constrained Delaunay triangulation that supports intersections of input constraints. This is the case when the template parameter *Itag* of (*Constrained_Delaunay_triangulation_2<Traits,Tds,Itag>*) is instantiated by one of the tag classes *Exact_intersections_tag* or *Exact_predicates_tag*). The concept ConstrainedDelaunayTriangulationTraits_2 refines both the concept *DelaunayTriangulationTraits_2* and the concept *ConstrainedTriangulationTraits_2*.

Refines

DelaunayTriangulationTraits_2
ConstrainedTriangulationTraits_2

Has Models

The kernels of CGAL are models for this traits class.
Triangulation_euclidean_traits_xy<K>
Triangulation_euclidean_traits_yz<K>
Triangulation_euclidean_traits_xz<K>

See Also

TriangulationTraits_2
ConstrainedTriangulationTraits_2
Constrained_triangulation_2<Gt,Tds,Itag>
Constrained_Delaunay_triangulation_2<Gt,Tds,Itag>

ConstrainedTriangulationFaceBase_2

Concept

Definition

In a constrained triangulation, the information about constrained edges is stored in the faces of the triangulation. The base face of a constrained triangulation has to be a model of the concept ConstrainedTriangulationFaceBase_2 which refines the concept *TriangulationFaceBase_2* providing functionalities to deal with constraints.

Refines

TriangulationFaceBase_2

Types

Defines the same types as the *TriangulationFaceBase_2* concept

Access Functions

bool *f.is_constrained(int i)*

returns true if the edge between f and its neighbor *f.neighbor(i)* is constrained. *Precondition*: $0 \leq i \leq 2$.

advanced

Modifiers

void *f.set_constraint(int i, bool b)*

sets the edge between f and its neighbor *f.neighbor(i)* as a constrained or unconstrained edge according to b.

void *f.set_constraints(bool c0, bool c1, bool c2)*

sets the status (constrained or unconstrained) of the three edges of f.

void	*f.reorient()*	Changes the orientation of f by exchanging *vertex(0)* with *vertex(1)* and *neighbor(0)* with *neighbor(1)* and the corresponding constrained status.
void	*f.ccw_permute()*	performs a counterclockwise permutation of the vertices, neighbors and constrained status of f.
void	*f.cw_permute()*	performs a clockwise permutation of the vertices and neighbors and constrained status of f.

Miscellaneous

bool	*f.is_valid()*	tests the validity of face f as a face of a plain triangulation and additionally checks if the edges of f are consistently marked as constrained or unconstrained edges in face f and its neighbors.

advanced

Has Models

CGAL::Constrained_triangulation_face_base_2<Traits>

See Also

TriangulationFaceBase_2
CGAL::Constrained_triangulation_2<Traits,Tds>
CGAL::Constrained_triangulation_face_base_2<Traits>

ConstrainedTriangulationTraits_2

Concept

Definition

The concept ConstrainedTriangulationTraits_2 defines the requirements for the geometric traits class of a constrained triangulation (*Constrained_Triangulation_2<Traits,Tds,Itag>*) that supports intersections of input constraints (i. e. when the template parameter *Itag* is instantiated by one of the tag classes *Exact_intersections_tag* or *Exact_predicates_tag*). This concept refines the concept *TriangulationTraits_2*, adding requirements for function objects to compute the intersection points of two constraints. When *Exact_predicates_tag* is used, the traits class is also required to provide additional types to compute the squared distance between a point and a line

Refines

TriangulationTraits_2

Types

ConstrainedTriangulationTraits_2:: Intersect_2

A function object whose operator() computes the intersection of two segments :
Object_2 operator()(Segment_2 s1, Segment_2 s2); Returns the intersection of *s1* and *s2*.

When the constrained triangulation is instantiated with the intersection tag *Exact_predicates_tag*, the used algorithm needs to be able to compare some distances between points and lines and the following types are further required.

ConstrainedTriangulationTraits_2:: RT A number type supporting the comparison operator <.

ConstrainedTriangulationTraits_2:: Line_2 The line type.

ConstrainedTriangulationTraits_2:: Construct_line_2

A function object whose operator() constructs a line from two points :
Line_2 operator()(Point_2 p1, Point_2 p2).

ConstrainedTriangulationTraits_2:: Compute_squared_distance_2

A function object with an operator() designed to compute the squared distance between a line and a point :
RT operator()(Line_2 l, Point_2 p); Return the squared distance between *p* and *l*.

Access to constructor object

Intersect_2 *traits.intersect_2_object()*

Construct_line_2 *traits.construct_line_2_object()*

required when the intersection tag is *Exact_predicates_tag*.

Compute_squared_distance_2

traits.compute_squared_distance_2_object()

required when the intersection tag is *Exact_predicates_tag*.

Has Models

The kernels of CGAL are models for this traits class.
CGAL::Projection_traits_xy_3<K> .... page 604
CGAL::Projection_traits_yz_3<K>
CGAL::Projection_traits_zx_3<K>

deprecated

CGAL::Triangulation_euclidean_traits_xy_3<K> .... page 2566,
CGAL::Triangulation_euclidean_traits_yz_3<K>,
CGAL::Triangulation_euclidean_traits_zx_3<K>.

deprecated

See Also

TriangulationTraits_2
ConstrainedDelaunayTriangulationTraits_2
CGAL:Constrained_Triangulation_2<Traits,Tds,Itag>

CGAL::Constrained_Delaunay_triangulation_2<Traits,Tds,Itag>

Definition

A constrained Delaunay triangulation is a triangulation with constrained edges which tries to be as much Delaunay as possible. Constrained edges are not necessarily Delaunay edges, therefore a constrained Delaunay triangulation is not a Delaunay triangulation. A constrained Delaunay is a triangulation whose faces do not necessarily fulfill the empty circle property but fulfill a weaker property called the *constrained empty circle*. To state this property, it is convenient to think of constrained edges as blocking the view. Then, a triangulation is constrained Delaunay if the circumscribing circle of any of its triangular faces includes in its interior no vertex that is visible from the interior of the triangle. The class *Constrained_Delaunay_triangulation_2<Traits,Tds,Itag>* is designed to represent constrained Delaunay triangulations.

As in the case of constrained triangulations, three different versions of Delaunay constrained triangulations are offered depending on whether the user wishes to handle intersecting input constraints or not. The desired version can be selected through the instantiation of the third template parameter *Itag* which can be one of the following :
CGAL::No_intersection_tag if intersections of input constraints are disallowed,
CGAL::Exact_predicates_tag allows intersections between input constraints and is to be used when the traits class provides exact predicates but approximate constructions of the intersection points,
CGAL::Exact_intersections_tag allows intersections between input constraints and is to be used in conjunction with an exact arithmetic type.

The template parameters *Tds* has to be instantiate with a model of *TriangulationDataStructure_2*. The geometric traits of a constrained Delaunay triangulation is required to provide the *side_of_oriented_circle* test as the geometric traits of a Delaunay triangulation and the *Traits* parameter has to be instantiated with a model *DelaunayTriangulationTraits_2*. When intersection of input constraints are supported, the geometric traits class is required to provide additional function object types to compute the intersection of two segments. and has then to be also a model of the concept *ConstrainedTriangulationTraits_2*.

A constrained Delaunay triangulation is not a Delaunay triangulation but it is a constrained triangulation. Therefore the class *Constrained_Delaunay_triangulation_2<Traits,Tds,Itag>* derives from the class *Constrained_triangulation_2<Traits,Tds>*. Also, information about the status (constrained or not) of the edges of the triangulation is stored in the faces. Thus the nested *Face* type of a constrained triangulation offers additional functionalities to deal with this information. These additional functionalities induce additional requirements on the base face class plugged into the triangulation data structure of a constrained Delaunay triangulation. The base face of a constrained Delaunay triangulation has to be a model of the concept *ConstrainedTriangulationFaceBase_2*.

CGAL provides a default for the template parameters. If *Gt* is the geometric traits parameter, the default for *ConstrainedTriangulationFaceBase_2* is the class *CGAL::Constrained_triangulation_face_base_2<Gt>* and the default for the triangulation data structure parameter is the class *CGAL::Triangulation_data_structure_2< CGAL::Triangulation_vertex_base_2<Gt>, CGAL::Constrained_triangulation_face_base_2<Gt> >*. The default intersection tag is *CGAL::No_intersection_tag*.

#include <CGAL/Constrained_Delaunay_triangulation_2.h>

Inherits From

Constrained_triangulation_2<Traits,Tds,Itag>

Types

All types used in this class are inherited from the base class *Constrained_triangulation_2<Traits,Tds,Itag>*.

Creation

Constrained_Delaunay_triangulation_2<Traits,Tds,Itag> cdt(Traits t = Traits());

Introduces an empty constrained Delaunay triangulation *cdt*.

Constrained_Delaunay_triangulation_2<Traits,Tds,Itag> cdt(Constrained_Delaunay_triangulation_2 cdt1);

Copy constructor, all faces and vertices are duplicated and the constrained status of edges is copied.

template<class InputIterator>
Constrained_Delaunay_triangulation_2<Traits,Tds,Itag> cdt(InputIterator first, InputIterator last, Traits t=Traits())

A templated constructor which introduces and builds a constrained triangulation with constrained edges in the range $[first, last)$.
Precondition: The *value_type* of *first* and *last* is *Constraint*.

Insertion and Removal

The following member functions overwrite the corresponding members of the base class to include a step restoring the Delaunay constrained property after modification of the triangulation.

Vertex_handle *cdt.insert(Point p, Face_handle f = Face_handle())*

Inserts point p in the triangulation. If present f is used as an hint for the location of p.

Vertex_handle *cdt.insert(Point p, Locate_type& lt, Face_handle loc, int li)*

Same as above except that the location of the point p to be inserted is assumed to be given by *(lt,loc,i)*.

Vertex_handle *cdt.push_back(Point p)*

Equivalent to *insert(p)*.

template < class InputIterator >
std::ptrdiff_t *cdt.insert(InputIterator first, InputIterator last)*

Inserts the points in the range $[first, last)$. Returns the number of inserted points.
Precondition: The *value_type* of *first* and *last* is *Point*.

void *cdt.insert_constraint(Point a, Point b)*

Inserts segment ab as a constrained edge in the triangulation.

void *cdt.push_back(Constraint c)*

Inserts constraints *c* as above.

void *cdt.insert_constraint(Vertex_handle va, Vertex_handle vb)*

Inserts the line segment whose endpoints are the vertices *va* and *vb* as a constrained edge e in the triangulation.

void *cdt.remove(Vertex_handle & v)*

Removes vertex v.
Precondition: Vertex *v* is not incident to a constrained edge.

void *cdt.remove_incident_constraints(Vertex_handle v)*

Make the edges incident to vertex *v* unconstrained edges.

void *cdt.remove_constraint(Face_handle f, int i)*

Edge *(f,i)* is no longer constrained.

Queries

The following template member functions query the set of faces in conflict with a point *p*. The notion of conflict refers here to a constrained Delaunay setting which means the following. Constrained edges are considered as visibility obstacles and a point *p* is considered to be in conflict with a face *f* iff it is visible from the interior of *f* and included in the circumcircle of *f*.

template <class OutputItFaces, class OutputItBoundaryEdges>
std::pair<OutputItFaces,OutputItBoundaryEdges>

cdt.get_conflicts_and_boundary(Point p,
OutputItFaces fit,
OutputItBoundaryEdges eit,
Face_handle start) const

OutItFaces is an output iterator with *Face_handle* as value type. *OutItBoundaryEdges* stands for an output iterator with *Edge* as value type. This members function outputs in the container pointed to by *fit* the faces which are in conflict with point *p*. It outputs in the container pointed to by *eit* the boundary of the zone in conflict with *p*. The boundary edges of the conflict zone are output in counterclockwise order and each edge is described through its incident face which is not in conflict with *p*. The function returns in a std::pair the resulting output iterators.
Precondition: dimension()==2

template <class OutputItFaces>
OutputItFaces *cdt.get_conflicts(Point p, OutputItFaces fit, Face_handle start) const*

Same as above except that only the faces in conflict with *p* are output. The function returns the resulting output iterator.
Precondition: dimension()==2

template <class OutputItBoundaryEdges>
OutputItBoundaryEdges

cdt.get_boundary_of_conflicts(Point p, OutputItBoundaryEdges eit, Face_handle start) const

OutputItBoundaryEdges stands for an output iterator with *Edge* as value type. This functions outputs in the container pointed to by *eit*, the boundary of the zone in conflict with *p*. The boundary edges of the conflict zone are output in counter-clockwise order and each edge is described through the incident face which is not in conflict with *p*. The function returns the resulting output iterator.

Checking

bool *cdt.is_valid()* Checks if the triangulation is valid and if each constrained edge is consistently marked constrained in its two incident faces.

advanced

Flips

bool *cdt.is_flipable(Face_handle f, int i)*

Determines if edge *(f,i)* can be flipped. Returns true if edge *(f,i)* is not constrained and the circle circumscribing f contains the vertex of *f->neighbor(i)* opposite to edge *(f,i)*.

void *cdt.flip(Face_handle& f, int i)*

Flip *f* and *f->neighbor(i)*.

void *cdt.propagating_flip(List_edges & edges)*

Makes the triangulation constrained Delaunay by flipping edges. List edges contains an initial list of edges to be flipped. The returned triangulation is constrained Delaunay if the initial list contains at least all the edges of the input triangulation that failed to be constrained Delaunay. (An edge is said to be constrained Delaunay if it is either constrained or locally Delaunay.)

advanced

See Also

CGAL::Constrained_triangulation_2<Traits,Tds,Itag>
TriangulationDataStructure_2
DelaunayTriangulationTraits_2
ConstrainedTriangulationTraits_2
ConstrainedDelaunayTriangulationTraits_2
ConstrainedTriangulationFaceBase_2

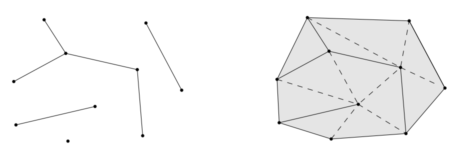
CGAL::Constrained_triangulation_2<Traits,Tds,Itag>

Definition

A constrained triangulation is a triangulation of a set of points which has to include among its edges a given set of segments joining the points. The given segments are called *constraints* and the corresponding edges in the triangulation are called *constrained edges*.

The endpoints of constrained edges are of course vertices of the triangulation. However the triangulation may include other vertices as well. There are three versions of constrained triangulations

- In the basic version, the constrained triangulation does not handle intersecting constraints, and the set of input constraints is required to be a set of segments that do not intersect except possibly at their endpoints. Any number of constrained edges are allowed to share the same endpoint. Vertical constrained edges are allowed as well as constrained edges with null length.
- The two other versions support intersecting input constraints. In those versions, input constraints are allowed to be intersecting, overlapping or partially overlapping segments. The triangulation introduce additional vertices at each point which is a proper intersection point of two constraints. A single constraint intersecting other constraints will then appear as the union of several constrained edges of the triangulation. The two versions dealing with intersecting constraints, slightly differ in the way intersecting constraints are dealt with.
  - One of them is designed to be robust when predicates are evaluated exactly but constructions (i. e. intersection computations) are approximate.
  - The other one is designed to be used with an exact arithmetic (meaning exact evaluation of predicates and exact computation of intersections.) This last version finds its full efficiency when used in conjunction with a constraint hierarchy data structure as provided in the class *Constrained_triangulation_plus_2*. See section 37.9.



The class *Constrained_triangulation_2<Traits,Tds,Itag>* of the CGAL library implements constrained triangulations. The template parameter *Traits* stands for a geometric traits class. It has to be a model of the concept *TriangulationTraits_2*. When intersection of input constraints are supported, the geometric traits class is required to provide additional function object types to compute the intersection of two segments. It has then to be a model of the concept *ConstrainedTriangulationTraits_2*. The template parameter *Tds* stands for a triangulation data structure class that has to be a model of the concept *TriangulationDataStructure_2*. The third parameter *Itag* is the intersection tag which serves to choose between the different strategies to deal with constraints intersections. CGAL provides three valid types for this parameter :
CGAL::No_intersection_tag disallows intersections of input constraints,
CGAL::Exact_predicates_tag is to be used when the traits class provides exact predicates but approximate constructions of the intersection points.
CGAL::Exact_intersections_tag is to be used in conjunction with an exact arithmetic type.

The information about constrained edges is stored in the faces of the triangulation. Thus the nested *Face* type of a constrained triangulation offers additional functionalities to deal with this information. These additional functionalities induce additional requirements on the base face class plugged into the triangulation data structure of a constrained Delaunay triangulation. The base face of a constrained Delaunay triangulation has to be a model of the concept *ConstrainedTriangulationFaceBase_2*.

CGAL provides default instantiations for the template parameters *Tds* and *Itag*, and for the *ConstrainedTriangulationFaceBase_2*. If *Gt* is the geometric traits parameter, the default for *ConstrainedTriangulationFaceBase_2* is the class *CGAL::Constrained_triangulation_face_base_2<Gt>* and the default for the triangulation data structure parameter is the class *CGAL::Triangulation_data_structure_2 < CGAL::Triangulation_vertex_base_2<Gt>, CGAL::Constrained_triangulation_face_base_2<Gt> >*. The default intersection tag is *CGAL::No_intersection_tag*.

#include <CGAL/Constrained_triangulation_2.h>

Inherits From

Triangulation_2<Traits,Tds>

Types

typedef std::pair<Point,Point>

Constraint; — The type of input constraints

typedef Itag *Intersection_tag;* — The intersection tag which decides how intersections between input constraints are dealt with.

Creation

Constrained_triangulation_2<Traits,Tds,Itag> ct;

default constructor.

Constrained_triangulation_2<Traits,Tds,Itag> ct(Constrained_triangulation_2 ct1);

Copy constructor, all faces and vertices are duplicated and the constrained status of edges is copied.

template<class InputIterator>
Constrained_triangulation_2<Traits,Tds,Itag> ct(InputIterator first, InputIterator last, Traits t=Traits());

A templated constructor which introduces and builds a constrained triangulation with constrained edges in the range $[first, last)$.
Precondition: The *value_type* of *first* and *last* is *Constraint*.

Queries

bool *ct.is_constrained(Edge e)*

Returns true if edge *e* is a constrained edge.

bool *ct.are_there_incident_constraints(Vertex_handle v)*

Returns true if at least one of the edges incident to vertex *v* is constrained.

template<class OutputItEdges>
OutputItEdges *ct.incident_constraints(Vertex_handle v, OutputItEdges out) const*

OutputItEdges is an output iterator with *Edge* as value type. Outputs the constrained edges incident to *v* in the sequence pointed to by *out* and returns the resulting output iterator.

Insertion and removal

Vertex_handle *ct.insert(Point p, Face_handle f = Face_handle())*

Inserts point *p* and restores the status (constrained or not) of all the touched edges. If present *f* is used as an hint for the location of *p*.

Vertex_handle *ct.insert(Point p, Locate_type& lt, Face_handle loc, int li)*

Same as above except that the location of the point *p* to be inserted is assumed to be given by *(lt,loc,i)*.

Vertex_handle *ct.push_back(Point p)*

Equivalent to *insert(p)*.

template < class InputIterator >
std::ptrdiff_t *ct.insert(InputIterator first, InputIterator last)*

Inserts the points in the range [*first*, *last*). Returns the number of inserted points.
Precondition: The *value_type* of *first* and *last* is *Point*.

void *ct.insert_constraint(Point a, Point b)*

Inserts points *a* and *b*, and inserts segment *ab* as a constraint. Removes the faces crossed by segment *ab* and creates new faces instead. If a vertex *c* lies on segment *ab*, constraint *ab* is replaced by the two constraints *ac* and *cb*. Apart from the insertion of *a* and *b*, the algorithm runs in time proportional to the number of removed triangles.
Precondition: The relative interior of segment *ab* does not intersect the relative interior of another constrained edge.

void *ct.push_back(Constraint c)*

Inserts constraints *c* as above.

void *ct.insert_constraint(Vertex_handle va, Vertex_handle vb)*

Inserts the line segment *s* whose endpoints are the vertices *va* and *vb* as a constrained edge *e*. The triangles intersected by *s* are removed and new ones are created.

void *ct.remove(Vertex_handle v)*

Removes a vertex *v*.
Precondition: Vertex *v* is not incident to a constrained edge.

void *ct.remove_incident_constraints(Vertex_handle v)*

Make the edges incident to vertex *v* unconstrained edges.

void *ct.remove_constrained_edge(Face_handle f, int i)*

Make edge *(f,i)* no longer constrained.

advanced

bool *ct.is_valid(bool verbose = false, int level = 0) const*

Checks the validity of the triangulation and the consistency of the constrained marks in edges.

advanced

I/O

ostream & *ostream& os << Constrained_triangulation_2<Traits,Tds> Ct*

Writes the triangulation as for *CGAL::Triangulation_2< Traits,Tds>* and, for each face f, and integers i=0,1,2, write “C” or “N” depending whether edge *(f,i)* is constrained or not.

istream& *istream& is >> Constrained_triangulation_2<Traits,Tds> Ct& t*

Reads a triangulation from stream *is* and assigns it to *t*. Data in the stream must have the same format *operator<<* uses. Note that *t* is first cleared.

See Also

CGAL::Triangulation_2<Traits,Tds>,
TriangulationDataStructure_2,
TriangulationTraits_2
ConstrainedTriangulationTraits_2
ConstrainedTriangulationFaceBase_2

Implementation

The insertion of a constrained edge runs in time proportional to the number of triangles intersected by this edge.

CGAL::Constrained_triangulation_face_base_2<Traits,Fb>

Class

Definition

The class *Constrained_triangulation_face_base_2<Traits,Fb>* is the default model for the concept *ConstrainedTriangulationFaceBase_2* to be used as base face class of constrained triangulations.

#include <CGAL/Constrained_triangulation_face_base_2.h>

Is Model for the Concepts

ConstrainedTriangulationFaceBase_2

Parameters

The first template parameter is a geometric traits.

The second template parameter has to be a model of the concept *TriangulationFaceBase_2*. Its default is *CGAL::Triangulation_face_base_2<Traits>*

Inherits From

The class *Constrained_triangulation_face_base_2<Traits,Fb>* derives from its parameter *Fb*. and add three Boolean to deal with information about constrained edges.

The member functions *cw(int i)*, *ccw(int i)* and *reorient* are overloaded to update information about constrained edges.

See Also

TriangulationFaceBase_2
ConstrainedTriangulationFaceBase_2
CGAL::Constrained_triangulation_2<Traits,Tds>
CGAL::Triangulation_face_base_2<Traits>

Class

CGAL::Constrained_triangulation_plus_2<Tr>

The class *Constrained_triangulation_plus_2<Tr>* implements a constrained triangulation with an additional data structure called the constraint hierarchy that keeps track of the input constraints and of their refinement in the triangulation.

The class *Constrained_triangulation_plus_2<Tr>* inherits from its template parameter Tr, which has to be instantiated by a constrained or constrained Delaunay triangulation.

According to its intersection tag, the base class will support intersecting input constraints or not. When intersections of input constraints are supported, the base class constructs a triangulation of the arrangement of the constraints, introducing new vertices at each proper intersection point and refining the input constraints into sub-constraints which are edges (more precisely constrained edges) of the triangulation. In this context, the constraint hierarchy keeps track of the input constraints and of their refinement in the triangulation. This data structure maintains for each input constraints the sequence of intersection vertices added on this constraint. The constraint hierarchy also allows the user to retrieve the set of constrained edges of the triangulation, and for each constrained edge, the set of input constraints that overlap it.

#include <CGAL/Constrained_triangulation_plus_2.h>

Inherits From

Tr which is either a *CGAL::Constrained_triangulation_2<Traits,Tds>* or a *CGAL::Constrained_Delaunay_triangulation_2<Traits,Tds>*.

Types

typedef Tr	*Triangulation;*	the triangulation base class.
typedef Itag	*Intersection_tag;*	the intersection tag.s

Constrained_triangulation_plus_2<Tr>:: Constraint_iterator

An iterator to visit all the input constraints. The order of visit is arbitrary. The value type of this iterator is a pair *std::pair<Vertex_handle, Vertex_handle>* corresponding to the endpoints of the constraint.

Constrained_triangulation_plus_2<Tr>:: Subconstraint_iterator

An iterator to visit all the sub-constraints of the triangulation. The order of visit is arbitrary. The value type of this iterator is a pair *std::pair<Vertex_handle, Vertex_handle>* corresponding to the vertices of the sub-constraint.

Constrained_triangulation_plus_2<Tr>:: Vertices_in_constraint_iterator

An iterator on the vertices of the chain of triangulation edges representing a constraint. The value type of this iterator is *Vertex_handle*.

typedef	*Context;*	This type is intended to describe a constraint enclosing a sub-constraint and the position of the sub-constraint in this constraint. It provides three member functions *vertices_begin()*, *vertices_end()* and *current()* returning iterators of the type *Vertices_in_constraint_iterator* on the sequence of vertices of the enclosing constraint. These iterators point respectively on the first vertex of the enclosing constraint, past the last vertex and on the first vertex of the sub-constraint.
typedef	*Context_iterator;*	An iterator on constraints enclosing a given sub-constraint. The value type of this iterator is *Context*.

Creation

Constrained_triangulation_plus_2<Tr> ctp(Geom_traits gt=Geom_traits());

Introduces an empty triangulation.

Constrained_triangulation_plus_2<Tr> ctp(Constrained_triangulation_plus_2 ct);

Copy constructor.

Constrained_triangulation_plus_2<Tr> ctp(std::list<Constraint>& lc, Geom_traits t = Geom_traits());

Introduces and builds a constrained triangulation from the list of constraints *lc*.

template<class InputIterator>
Constrained_triangulation_plus_2<Tr> ctp(InputIterator first,
InputIterator last,
Geom_traits gt= Geom_traits())

Introduces and builds a constrained triangulation from the constraints in the range $[first, last)$.
Precondition: The *value_type* of *first* and *last* is *Constraint*.

Assignment

Constrained_triangulation_plus_2

ctp = Constrained_triangulation_plus_2 tr

Assignment. All the vertices and faces are duplicated. The constraint hierarchy is also duplicated.

void *ctp.swap(Constrained_triangulation_plus_2 tr)*

The triangulations *tr* and *ctp* are swapped. This operation should be preferred to *ctp = tr* or to *t(tr)* if *tr* is deleted after that.

Insertion and Removal

The class *Constrained_triangulation_plus_2<Tr>* overwrites the following insertion and removal member functions for points and constraints.

Vertex_handle *ctp.insert(Point p, Face_handle start = Face_handle())*

Inserts point *p* as a vertex of the triangulation.

Vertex_handle *ctp.insert(Point p, Locate_type lt, Face_handle loc, int li)*

inserts a point *p* whose location is assumed to be given by *(lt,loc,li)*.

Vertex_handle *ctp.push_back(Point p)*

Equivalent to *insert(p)*.

template < class InputIterator >
size_type *ctp.insert(InputIterator first, InputIterator last)*

Inserts the points in the range $[first, last)$. Returns the number of inserted points.
Precondition: The *value_type* of *first* and *last* is *Point*.

void *ctp.insert_constraint(Point a, Point b)*

Inserts the constraint segment ab in the triangulation.

void *ctp.push_back(Constraint c)*

Inserts the constraint *c*.

void *ctp.insert_constraint(Vertex_handle va, Vertex_handle vb)*

Inserts a constraint whose endpoints are the vertices pointed by *va* and *vb* in the triangulation.

void *ctp.remove_constraint(Vertex_handle va, Vertex_handle vb)*

Removes the constraint joining the vertices pointed by *va* and *vb*.
Precondition: *va* and *vb* have to refer to the endpoint vertices of an input constraint.

Queries

Constraint_iterator *ctp.constraints_begin() const*

Returns a *Constraint_iterator* pointing on the first constraint.

Constraint_iterator *ctp.constraints_end() const*

Returns a *Constraint_iterator* pointing past the last constraint.

Subconstraint_iterator

ctp.subconstraints_begin() const

Returns a *Subconstraint_iterator* pointing on the first sub-constraint.

Subconstraint_iterator

ctp.subconstraints_end() const

Returns a *Subconstraint_iterator* pointing past the last sub-constraint.

int *ctp.number_of_enclosing_constraints(Vertex_handle va, Vertex_handle vb)*

Returns the number of constraints enclosing the sub-constraint *(va,vb)*.
Precondition: *va* and *vb* refer to the vertices of a constrained edge of the triangulation.

Context *ctp.context(Vertex_handle va, Vertex_handle vb)*

Returns the *Context* relative to one of the constraint enclosing the sub-constraint *(va,vb)*.
Precondition: *va* and *vb* refer to the vertices of a constrained edge of the triangulation.

Context_iterator *ctp.contexts_begin(Vertex_handle va, Vertex_handle vb)*

Returns an iterator pointing on the first *Context* of the sequence of *Contexts* corresponding to the constraints enclosing the sub-constraint*(va,vb)*.
Precondition: *va* and *vb* refer to the vertices of a constrained edge of the triangulation.

Context_iterator *ctp.contexts_end(Vertex_handle va, Vertex_handle vb)*

Returns an iterator past the last *Context* of the sequence of *Contexts* corresponding to the constraints enclosing the *(va,vb)*.
Precondition: *va* and *vb* refer to the vertices of a constrained edge of the triangulation.

Vertices_in_constraint_iterator

ctp.vertices_in_constraint_begin(Vertex_handle va, Vertex_handle vb)

Returns an iterator on the first vertex on the constraint *(va,vb)*
Precondition: *va* and *vb* refer to the vertices of an input constraint.

Vertices_in_constraint_iterator

ctp.vertices_in_constraint_end(Vertex_handle va, Vertex_handle vb)

Returns an iterator past the last vertex on the constraint *(va,vb)*
Precondition: *va* and *vb* refer to the vertices of an input constraints.

See Also

CGAL::Constrained_triangulation_2<Traits,Tds>
CGAL::Constrained_Delaunay_triangulation_2<Traits,Tds>
ConstrainedTriangulationTraits_2
ConstrainedDelaunayTriangulationTraits_2

DelaunayTriangulationTraits_2

Definition

In addition to the requirements of the concept *TriangulationTraits_2* described page 2541, the concept DelaunayTriangulationTraits_2 provide a predicate to check the empty circle property. The corresponding predicate type is called type *Side_of_oriented_circle_2*.

The additional types *Line_2*, *Ray_2* and the constructor objects *Construct_ray_2 Construct_circumcenter_2*, *Construct_bisector_2*, *Construct_midpoint* are used to build the dual Voronoi diagram and are required only if the dual functions are called. The additional predicate type *Compare_distance_2* is required if calls to *nearest_vertex(..)* are issued.

Refines

TriangulationTraits_2

Types

DelaunayTriangulationTraits_2:: Line_2 — The line type. This type is required only if some dual functions are called.

DelaunayTriangulationTraits_2:: Ray_2 — The type for ray. This type is required only if some dual functions are called.

DelaunayTriangulationTraits_2:: Side_of_oriented_circle_2

Predicate type. Provides the operator :
Oriented_side operator()(Point p, Point q, Point r, Point s) which takes four points p,q,r,s as arguments and returns *ON_POSITIVE_SIDE*, *ON_NEGATIVE_SIDE* or, *ON_ORIENTED_BOUNDARY* according to the position of points s with respect to the oriented circle through p,q and r.

DelaunayTriangulationTraits_2:: Compare_distance_2

Predicate type. Provides the operator :
Comparison_result operator()(Point_2 p, Point_2 q, Point_2 r) which returns *SMALLER*, *EQUAL* or *LARGER* according to the distance between p and q being smaller, equal or larger than the distance between p and r. This type is only require if *nearest_vertex* queries are issued.

DelaunayTriangulationTraits_2:: Construct_circumcenter_2

Constructor object. Provides the operator :
Point_2 operator()(Point_2 p, Point_2 q, Point_2 r) which returns the circumcenter of the three points *p, q* and *r*. This type is required only if functions relative to the dual Voronoi diagram are called.

DelaunayTriangulationTraits_2:: Construct_bisector_2

Constructor object. Provides the operator : *Line_2 operator()(Point_2 p, Point_2 q)* which constructs the bisector line of points p and q. This type is required only if functions relative to the dual Voronoi diagram are called.

DelaunayTriangulationTraits_2:: Construct_ray_2

A constructor object to build a ray from a point and a line. Provides :
Ray_2 operator() (Point_2 p, Line_2 l);

Creation

DelaunayTriangulationTraits_2 traits;	default constructor.
DelaunayTriangulationTraits_2 traits(dtt);	copy constructor
DelaunayTriangulationTraits_2 traits = traits2	Assignment operator.

Access to predicate and constructor objects

Side_of_oriented_circle_2 *traits.side_of_oriented_circle_2_object()*

The following functions are required only if member functions of the Delaunay triangulation relative to the dual Voronoi diagram are called.

Compare_distance_2	*traits.compare_distance_2_object()*
Construct_circumcenter_2	*traits.construct_circumcenter_2_object()*
Construct_bisector_2	*traits.construct_bisector_2_object()*
Construct_direction_2	*traits.construct_direction_2_object()*
Construct_ray_2	*traits.construct_ray_2_object()*

Has Models

CGAL kernels
CGAL::Triangulation_euclidean_traits_2<Rep>.

The following traits classes provide everything except types and member functions required for the dual Voronoi diagram:
CGAL::Projection_traits_xy_3<K> . . . . page 604
CGAL::Projection_traits_yz_3<K>
CGAL::Projection_traits_zx_3<K>

deprecated

CGAL::Triangulation_euclidean_traits_xy_3<K> . . . . page 2566,
CGAL::Triangulation_euclidean_traits_yz_3<K>,
CGAL::Triangulation_euclidean_traits_zx_3<K>.

deprecated

See Also

TriangulationTraits_2

Class

CGAL::Delaunay_triangulation_2<Traits,Tds>

Definition

The class *Delaunay_triangulation_2<Traits,Tds>* is designed to represent the Delaunay triangulation of a set of points in a plane. A Delaunay triangulation of a set of points is a triangulation of the sets of points that fulfills the following *empty circle property* (also called *Delaunay property*): the circumscribing circle of any facet of the triangulation contains no point of the set in its interior. For a point set with no case of co-circularity of more than three points, the Delaunay triangulation is unique, it is the dual of the Voronoi diagram of the points.

#include <CGAL/Delaunay_triangulation_2.h>

Parameters

The template parameter *Tds* is to be instantiated with a model of *TriangulationDataStructure_2*. CGAL provides a default instantiation for this parameter, which is the class *CGAL::Triangulation_data_structure_2 < CGAL::Triangulation_vertex_base_2<Traits>, CGAL::Triangulation_face_base_2<Traits> >*.

The geometric traits *Traits* is to be instantiated with a model of *DelaunayTriangulationTraits_2*. The concept *DelaunayTriangulationTraits_2* refines the concept *TriangulationTraits_2*, providing a predicate type to check the empty circle property.

Changing this predicate type allows to build Delaunay triangulations for different metrics such that L_1 or L_∞ or any metric defined by a convex object. However, the user of an exotic metric must be careful that the constructed triangulation has to be a triangulation of the convex hull which means that convex hull edges have to be Delaunay edges. This is granted for any smooth convex metric (like L_2) and can be ensured for other metrics (like L_∞) by the addition to the point set of well chosen sentinel points The concept of *DelaunayTriangulationTraits_2* is described page 2513.

When dealing with a large triangulations, the user is advised to encapsulate the Delaunay triangulation class into a triangulation hierarchy, which means to use the class *Triangulation_hierarchy_2<Tr>* with the template parameter instantiated with *Delaunay_triangulation_2<Traits,Tds>* . The triangulation hierarchy will then offer the same functionalities but be much more for efficient for locations and insertions.

Inherits From

Triangulation_2<Traits,Tds>

Types

Inherits all the types defined in *Triangulation_2<Traits,Tds>*.

Creation

Delaunay_triangulation_2<Traits,Tds> dt(Traits gt = Traits());

default constructor.

Delaunay_triangulation_2<Traits,Tds> dt(tr);

copy constructor. All the vertices and faces are duplicated.

Insertion and Removal

The following insertion and removal functions overwrite the functions inherited from the class *Triangulation_2<Traits,Tds>* to maintain the Delaunay property.

In the degenerate case when there are co-circular points, the Delaunay triangulation is known not to be uniquely defined. In this case, CGAL chooses a particular Delaunay triangulation using a symbolic perturbation scheme [DT03].

Vertex_handle *dt.insert(Point p, Face_handle f=Face_handle())*

inserts point *p*. If point *p* coincides with an already existing vertex, this vertex is returned and the triangulation is not updated. Optional parameter *f* is used to initialize the location of *p*.

Vertex_handle *dt.insert(Point p, Locate_type& lt, Face_handle loc, int li)*

inserts a point *p*, the location of which is supposed to be given by *(lt,loc,li)*, see the description of member function *locate* in class *Triangulation_2<Traits,Tds>*.

Vertex_handle *dt.push_back(Point p)*

equivalent to *insert(p)*.

template < class PointInputIterator >
std::ptrdiff_t *dt.insert(PointInputIterator first, PointInputIterator last)*

inserts the points in the range $[first, last)$. Returns the number of inserted points. Note that this function is not guaranteed to insert the points following the order of *PointInputIterator*, as *spatial_sort* is used to improve efficiency.
Precondition: The *value_type* of *first* and *last* is *Point*.

template < class PointWithInfoInputIterator >

std::ptrdiff_t *dt.insert(PointWithInfoInputIterator first, PointWithInfoInputIterator last)*

inserts the points in the iterator range $[first, last)$. Returns the number of inserted points. Note that this function is not guaranteed to insert the points following the order of *PointWithInfoInputIterator*, as *spatial_sort* is used to improve efficiency. Given a pair *(p,i)*, the vertex *v* storing *p* also stores *i*, that is *v.point() == p* and *v.info() == i*. If several pairs have the same point, only one vertex is created, and one of the objects of type *Vertex::Info* will be stored in the vertex.
Precondition: *Vertex* must be model of the concept *TriangulationVertexBaseWithInfo_2*. The *value_type* of *first* and *last* is *std::pair<Point,Vertex::Info>*.

void *dt.remove(Vertex_handle v)*

removes the vertex from the triangulation.

Note that the other modifier functions of *Triangulation_2<Traits,Tds>* are not overwritten. Thus a call to *insert_in_face insert_in_edge*, *insert_outside_convex_hull*, *insert_outside_affine_hull* or *flip* on a valid Delaunay triangulation might lead to a triangulation which is no longer a Delaunay triangulation.

Displacement

Vertex_handle *dt.move_if_no_collision(Vertex_handle v, Point p)*

if there is not already another vertex placed on *p*, the triangulation is modified such that the new position of vertex *v* is *p*, and *v* is returned. Otherwise, the triangulation is not modified and the vertex at point *p* is returned.
Precondition: Vertex *v* must be finite.

Vertex_handle *dt.move(Vertex_handle v, Point p)*

same as above if there is no collision. Otherwise, *v* is deleted and the vertex placed on *p* is returned.
Precondition: Vertex *v* must be finite.

Queries

Vertex_handle *dt.nearest_vertex(Point p, Face_handle f=Face_handle())*

returns any nearest vertex of *p*. The implemented function begins with a location step and *f* may be used to initialize the location.

template <class OutputItFaces, class OutputItBoundaryEdges>

std::pair<OutputItFaces,OutputItBoundaryEdges>

dt.get_conflicts_and_boundary(Point p,
OutputItFaces fit,
OutputItBoundaryEdges eit,
Face_handle start) const

OutputItFaces is an output iterator with *Face_handle* as value type. *OutputItBoundaryEdges* stands for an output iterator with *Edge* as value type. This members function outputs in the container pointed to by *fit* the faces which are in conflict with point *p* i. e. the faces whose circumcircle contains *p*. It outputs in the container pointed to by *eit* the the boundary of the zone in conflict with *p*. The boundary edges of the conflict zone are output in counter-clockwise order and each edge is described through its incident face which is not in conflict with *p*. The function returns in a std::pair the resulting output iterators.
Precondition: dimension()==2

template <class OutputItFaces>
OutputItFaces *dt.get_conflicts(Point p, OutputItFaces fit, Face_handle start) const*

same as above except that only the faces in conflict with *p* are output. The function returns the resulting output iterator.
Precondition: dimension()==2

template <class OutputItBoundaryEdges>
OutputItBoundaryEdges

dt.get_boundary_of_conflicts(Point p,
OutputItBoundaryEdges eit,
Face_handle start) const

OutputItBoundaryEdges stands for an output iterator with *Edge* as value type. This function outputs in the container pointed to by *eit*, the boundary of the zone in conflict with *p*. The boundary edges of the conflict zone are output in counterclockwise order and each edge is described through the incident face which is not in conflict with *p*. The function returns the resulting output iterator.

Voronoi diagram

The following member functions provide the elements of the dual Voronoi diagram.

Point *dt.dual(Face_handle f) const*

Returns the center of the circle circumscribed to face f.
Precondition: f is not infinite

Object *dt.dual(Edge e) const* returns a segment, a ray or a line supported by the bisector of the endpoints of *e*. If faces incident to *e* are both finite, a segment whose endpoints are the duals of each incident face is returned. If only one incident face is finite, a ray whose endpoint is the dual of the finite incident face is returned. Otherwise both incident faces are infinite and the bisector line is returned.

Object *dt.dual(Edge_circulator ec) const*

Idem

Object *dt.dual(Edge_iterator ei) const*

Idem

template < class Stream>
Stream&

dt.draw_dual(Stream & ps) output the dual Voronoi diagram to stream *ps*.

Predicates

Oriented_side *dt.side_of_oriented_circle(Face_handle f, Point p) const*

Returns the side of *p* with respect to the circle circumscribing the triangle associated with *f*

advanced

Miscellaneous

The checking function *is_valid()* is also overwritten to additionally test the empty circle property.

bool *dt.is_valid(bool verbose = false, int level = 0) const*

tests the validity of the triangulation as a *Triangulation_2* and additionally tests the Delaunay property. This method is mainly useful for debugging Delaunay triangulation algorithms designed by the user.

advanced

See Also

CGAL::Triangulation_2<Traits,Tds>,
TriangulationDataStructure_2,
DelaunayTriangulationTraits_2,
Triangulation_hierarchy_2<Tr>.

Implementation

Insertion is implemented by inserting in the triangulation, then performing a sequence of Delaunay flips. The number of flips is $O(d)$ if the new vertex is of degree d in the new triangulation. For points distributed uniformly at random, insertion takes time $O(1)$ on average.

Removal calls the removal in the triangulation and then re-triangulates the hole in such a way that the Delaunay criterion is satisfied. Removal of a vertex of degree d takes time $O(d^2)$. The degree d is $O(1)$ for a random vertex in the triangulation.

After a point location step, the nearest neighbor is found in time $O(n)$ in the worst case, but in time $O(1)$ for vertices distributed uniformly at random and any query point.

Enum

CGAL::Triangulation_2<Traits,Tds>::Locate_type

Definition

The enum *Locate_type* is defined by the *Triangulation_2<Traits,Tds>* class to specify which case occurs when locating a point in the triangulation.

enum Locate_type { *VERTEX=0, EDGE, FACET, OUTSIDE_CONVEX_HULL, OUTSIDE_AFFINE_HULL*};

The locate type is :
VERTEX when the located point coincides with a vertex of the triangulation
EDGE when the point is in the relative interior of an edge
FACE when the point is in the interior of a facet
OUTSIDE_CONVEX_HULL when the point is outside the convex hull but in the affine hull of the current triangulation
OUTSIDE_AFFINE_HULL when the point is outside the affine hull of the current triangulation.

See Also

CGAL::Triangulation_2<Traits,Tds>.

RegularTriangulationFaceBase_2

Definition

The regular triangulation of a set of weighted points does not necessarily have one vertex for each of the input points. Some of the input weighted points have no cell in the dual power diagrams and therefore do not correspond to a vertex of the regular triangulation. Those weighted points are said to be *hidden* points. A point which is hidden at a given time may appear later as a vertex of the regular triangulation upon removal on some other weighted point. Therefore, hidden points have to be stored somewhere. The regular triangulation store those hidden points in special vertices called *hidden* vertices.

A hidden point can appear as vertex of the triangulation only when the two dimensional face where its point component is located (the face which hides it) is removed. Therefore we decided to store in each face of a regular triangulation the list of hidden vertices whose points are located in the face. Thus points hidden by a face are easily reinserted in the triangulation when the face is removed.

The base face of a regular triangulation has to be a model of the concept RegularTriangulationFaceBase_2 , which refines the concept *TriangulationFaceBase_2* by adding in the face a list to store hidden vertices.

Refines

TriangulationFaceBase_2

Types

typedef std::list<Vertex_handle>

	Vertex_list;	An std list of hidden vertices.

Access Functions

Vertex_list&	*rfb.vertex_list()*	Returns a reference to the list of vertices hidden by the face.

Has Models

CGAL::Regular_triangulation_face_base_2

See Also

TriangulationFaceBase_2
RegularTriangulationVertexBase_2

RegularTriangulationTraits_2

Definition

The concept RegularTriangulationTraits_2 describe the requirements for the traits class of regular triangulations. It refines the concept *TriangulationTraits_2* providing the type *Weighted_point_2* and the *power-test* predicate on those weighted points. A weighted point is basically a point augmented with a scalar weight. It can be seen as a circle when the weight is interpreted as a square radius. The *power-test* on weighted points is the fundamental test to build regular triangulations as the *side_of_oriented_circle* test is the fundamental test of Delaunay triangulations.

Refines

TriangulationTraits_2

Types

RegularTriangulationTraits_2:: Bare_point — Another name for the point type.

RegularTriangulationTraits_2:: Weighted_point_2

The weighted point type, it has to be a model of the concept *WeightedPoint*.

RegularTriangulationTraits_2:: Power_test_2 — A predicate object type. Must provide the operators:
— *Oriented_side operator() (Weighted_point_2 p, Weighted_point_2 q, Weighted_point_2 r, Weighted_point_2 s)* which is the power test for points *p*, *q*, *r* and *s*.
Precondition: the bare points corresponding to *p*, *q*, *r* are not collinear.
— *Oriented_side operator() (Weighted_point_2 p, Weighted_point_2 q, Weighted_point_2 r)* which is the degenerated power test for collinear points *p*, *q*, *r*.
Precondition: the bare points corresponding to *p*, *q*, *r* are collinear and *p != q*.
— *Oriented_side operator() (Weighted_point_2 p, Weighted_point_2 q)* which is the degenerated power test for weighted points *p* and *q* whose corresponding bare-points are identical.
Precondition: the bare points corresponding to *p* and *q* are identical.

The following type/predicate is required if a call to *nearest_power_vertex* is issued:

RegularTriangulationTraits_2:: Compare_power_distance_2

A predicate object type. Must provide the operator:
Comparison_result operator()(Bare_point p, Weighted_point_2 q, Weighted_point_2 r), which compares the power distance between *p* and *q* to the power distance between *p* and *r*.

RegularTriangulationTraits_2:: Construct_weighted_circumcenter_2

A constructor object which constructs the weighted circumcenter of three weighted points. Provides the operator *Bare_point operator() (Weighted_point_2 p, Weighted_point_2 q, Weighted_point_2 r);*

RegularTriangulationTraits_2:: Construct_radical_axis_2

A constructor type which constructs the radical axis of two weighted points. Provides the operator : *Line_2 operator() (Weighted_point_2 p, Weighted_point_2 q);*

Creation

RegularTriangulationTraits_2 traits; default constructor.

RegularTriangulationTraits_2 traits(RegularTriangulatioTraits_2);

copy constructor.

RegularTriangulationTraits_2&

traits = RegularTriangulationTraits_2

assignment operator

Access to predicate and constructors objects

Power_test_2	*traits.power_test_2_object()*
Compare_power_distance_2	*traits.compare_power_distance_2_object()*
Construct_weighted_circumcenter_2	*traits.construct_weighted_circumcenter_2_object()*
Construct_radical_axis_2	*traits.construct_radical_axis_2_object()*

Has Models

CGAL::Regular_triangulation_traits_2<Rep>
CGAL::Regular_triangulation_filtered_traits_2<FK>

See Also

TriangulationTraits_2
CGAL::Regular_triangulation_2<Traits,Tds>

RegularTriangulationVertexBase_2

Definition

The regular triangulation of a set of weighted points does not necessarily have one vertex for each of the input points. Some of the input weighted points have no cell in the dual power diagrams and therefore do not correspond to a vertex of the regular triangulation. Those weighted point are said to be *hidden* points. A point which is hidden at a given time may appear later as a vertex of the regular triangulation upon removal on some other weighted point. Therefore, hidden points have to be stored somewhere. The regular triangulation store those hidden points in special vertices called *hidden* vertices.

A hidden point can appear as vertex of the triangulation only when the two dimensional face where its point component is located (the face which hides it) is removed. Therefore we decided to store in each face of a regular triangulation the list of hidden vertices whose points are located in the face. Thus points hidden by a face are easily reinserted in the triangulation when the face is removed.

The base vertex of a regular triangulation has to be a model of the concept RegularTriangulationVertexBase_2 . The concept RegularTriangulationVertexBase_2 refines the concept *TriangulationVertexBase_2*, just adding a Boolean to mark if the vertex is a vertex of the triangulation or a hidden vertex.

Refines

TriangulationVertexBase_2

Access Functions

bool	*rvb.is_hidden()*	returns *true*, iff the vertex is hidden.
void	*rvb.set_hidden(bool b)*	Mark the vertex as hidden or as not hidden.

Has Models

CGAL::Regular_triangulation_vertex_base_2

See Also

TriangulationVertexBase_2 CGAL::Regular_triangulation_vertex_base_2

CGAL::Regular_triangulation_2<Traits,Tds>

Definition

The class *Regular_triangulation_2<Traits,Tds>* is designed to maintain the regular triangulation of a set of weighted points.

Let $PW = \{(p_i, w_i), i = 1, \ldots, n\}$ be a set of weighted points where each p_i is a point and each w_i is a scalar called the weight of point p_i. Alternatively, each weighted point (p_i, w_i) can be regarded as a two dimensional sphere with center p_i and radius $r_i = \sqrt{w_i}$.

The power diagram of the set PW is a planar partition such that each cell corresponds to sphere (p_i, w_i) of PW and is the locus of points p whose power with respect to (p_i, w_i) is less than its power with respect to any other sphere (p_j, w_j) in PW. The dual of this diagram is a triangulation whose domain covers the convex hull of the set $P = \{p_i, i = 1, \ldots, n\}$ of center points and whose vertices are a subset of P. Such a triangulation is called a regular triangulation. The three points p_i, p_j and p_k of P form a triangle in the regular triangulation of PW iff there is a point p of the plane whose powers with respect to (p_i, w_i), (p_j, w_j) and (p_k, w_k) are equal and less than the power of p with respect to any other sphere in PW.

Let us defined the power product of two weighted points (p_i, w_i) and (p_j, w_j) as:

$$\Pi(p_i, w_i, p_j, w_j) = p_i p_j^2 - w_i - w_j.$$

$\Pi(p_i, w_i, p_j, 0)$ is simply the power of point p_j with respect to the sphere (p_i, w_i), and two weighted points are said to be orthogonal if their power product is null. The power circle of three weighted points (p_i, w_i), (p_j, w_j) and (p_k, w_k) is defined as the unique circle (π, ω) orthogonal to (p_i, w_i), (p_j, w_j) and (p_k, w_k).

The regular triangulation of the sets PW satisfies the following *regular property* (which just reduces to the Delaunay property when all the weights are null): a triangle $p_i p_j p_k$ of the regular triangulation of PW is such that the power product of any weighted point (p_l, w_l) of PW with the power circle of (p_i, w_i), (p_j, w_j) is (p_k, w_k) is positive or null. We call power test of the weighted point (p_l, w_l) with respect to the face $p_i p_j p_k$, the predicates testing the sign of the power product of (p_l, w_l) with respect to the power circle of (p_i, w_i), (p_j, w_j) is (p_k, w_k). This power product is given by the following determinant

$$\begin{vmatrix} 1 & x_i & y_i & x_i^2 + y_i^2 - w_i \\ 1 & x_j & y_j & x_j^2 + y_j^2 - w_j \\ 1 & x_k & y_k & x_k^2 + y_k^2 - w_k \\ 1 & x_l & y_l & x_l^2 + y_l^2 - w_l \end{vmatrix}$$

A pair of neighboring faces $p_i p_j p_k$ and $p_i p_j p_l$ is said to be locally regular (with respect to the weights in PW) if the power test of (p_l, w_l) with respect to $p_i p_j p_k$ is positive. A classical result of computational geometry establishes that a triangulation of the convex hull of P such that any pair of neighboring faces is regular with respect to PW, is a regular triangulation of PW.

Alternatively, the regular triangulation of the weighted points set PW can be obtained as the projection on the two dimensional plane of the convex hull of the set of three dimensional points $P' = \{(p_i, p_i^2 - w_i), i = 1, \ldots, n\}$.

The vertices of the regular triangulation of a set of weighted points PW form only a subset of the set of center points of PW. Therefore the insertion of a weighted point in a regular triangulation does not necessarily imply the creation of a new vertex. If the new inserted point does not appear as a vertex in the regular triangulation, it is said to be hidden.

Hidden points are stored in special vertices called hidden vertices. A hidden point is considered as hidden by the facet of the triangulation where its point component is located : in fact, the hidden point can appear as vertex of the triangulation only if this facet is removed. Each face of a regular triangulation stores the list of hidden vertices whose points are located in the facet. When a facet is removed, points hidden by this facet are reinserted in the triangulation.

#include <CGAL/Regular_triangulation_2.h>

Parameters

The geometric traits parameter *Traits* has to be instantiated with a model of the concept *RegularTriangulationTraits_2*. The concept *RegularTriangulationTraits_2* refines the concept *TriangulationTraits_2* by adding the type *Weighted_point_2* to describe weighted points and the type *Power_test_2* to perform power tests on weighted points.

The *Tds* parameter has to be instantiated by a model of *TriangulationDataStructure_2*. The face base of a regular triangulation has to be a model of the concept *RegularTriangulationFaceBase_2*. while the vertex base class has to be a model of *RegularTriangulationVertexBase_2*. CGAL provides a default instantiation for the *Tds* parameter by the class *CGAL::Triangulation_data_structure_2 < CGAL::Reugular_triangulation_vertex_base_2<Traits>, CGAL::Regular_Triangulation_face_base_2<Traits> >*.

Inherits From

Triangulation_2<Traits,Tds>

Types

typedef Traits::Distance	*Distance;*
typedef Traits::Line	*Line;*
typedef Traits::Ray	*Ray;*
typedef Traits::Bare_point	*Bare_point;*
typedef Traits::Weighted_point	*Weighted_point;*

Regular_triangulation_2<Traits,Tds>:: All_vertices_iterator

An iterator that allows to enumerate the vertices that are not hidden.

Regular_triangulation_2<Traits,Tds>:: Finite_vertices_iterator

An iterator that allows to enumerate the finite vertices that are not hidden.

Regular_triangulation_2<Traits,Tds>:: Hidden_vertices_iterator

An iterator that allows to enumerate the hidden vertices.

Creation

Regular_triangulation_2<Traits,Tds> rt(Traits gt = Traits());

Introduces an empty regular triangulation *rt*.

Regular_triangulation_2<Traits,Tds> rt(Regular_triangulation_2 rt);

Copy constructor.

Insertion and Removal

Vertex_handle *rt.insert(Weighted_point p, Face_handle f=Face_handle())*

inserts weighted point *p* in the regular triangulation. If the point *p* does not appear as a vertex of the triangulation, the returned vertex is a hidden vertex. If given the parameter *f* is used as an hint for the place to start the location process of point *p*.

Vertex_handle *rt.insert(Weighted_point p, Locate_type lt, Face_handle loc, int li)*

insert a weighted point *p* whose bare-point is assumed to be located in *lt,loc,li*.

Vertex_handle *rt.push_back(Point p)* Equivalent to *insert(p)*.

template < class InputIterator >
std::ptrdiff_t *rt.insert(InputIterator first, InputIterator last)*

inserts the weighted points in the range $[first, last)$. It returns the difference of the number of vertices between after and before the insertions (it may be negative due to hidden points). Note that this function is not guaranteed to insert the weighted points following the order of *InputIterator*, as *spatial_sort* is used to improve efficiency.
Precondition: The *value_type* of *first* and *last* is *Weighted_point*.

template < class WeightedPointWithInfoInputIterator >
std::ptrdiff_t *rt.insert(WeightedPointWithInfoInputIterator first,*

WeightedPointWithInfoInputIterator last)

inserts the weighted points in the iterator range [*first*, *last*). It returns the difference of the number of vertices between after and before the insertions (it may be negative due to hidden points). Note that this function is not guaranteed to insert the weighted points following the order of *WeightedPointWithInfoInputIterator*, as *spatial_sort* is used to improve efficiency. Given a pair *(p,i)*, the vertex *v* storing *p* also stores *i*, that is *v.point() == p* and *v.info() == i*. If several pairs have the same point, only one vertex is created, one of the objects of type *Vertex::Info* will be stored in the vertex.
Precondition: *Vertex* must be model of the concept *TriangulationVertexBaseWithInfo_2*. The *value_type* of *first* and *last* is *std::pair<Weighted_point,Vertex::Info>*.

void *rt.remove(Vertex_handle v)* removes the vertex from the triangulation.

Queries

template <class OutputItFaces, class OutputItBoundaryEdges, class OutputItHiddenVertices>
CGAL::Triple<OutputItFaces,OutputItBoundaryEdges,OutputItHiddenVertices>

rt.get_conflicts_and_boundary_and_hidden_vertices(Weighted_point p,
OutputItFaces fit,
OutputItBoundaryEdges eit,
OutputItHiddenVertices vit,
Face_handle start) const

OutputItFaces is an output iterator with *Face_handle* as value type. *OutputItBoundaryEdges* stands for an output iterator with *Edge* as value type. *OutputItHiddenVertices* is an output iterator with *Vertex_handle* as value type. This member function outputs in the container pointed to by *fit* the faces which are in conflict with point *p* i. e. the faces whose power circles have negative power wrt. *p*. It outputs in the container pointed to by *eit* the boundary of the zone in conflict with *p*. It inserts the vertices that would be hidden by *p* into the container pointed to by *vit*. The boundary edges of the conflict zone are output in counter-clockwise order and each edge is described through its incident face which is not in conflict with *p*. The function returns in a CGAL::Triple the resulting output iterators.
Precondition: dimension()==2

template <class OutputItFaces, class OutputItBoundaryEdges>
std::pair<OutputItFaces,OutputItBoundaryEdges>

rt.get_conflicts_and_boundary(Weighted_point p,
OutputItFaces fit,
OutputItBoundaryEdges eit,

Face_handle start) const

same as above except that only the faces in conflict with *p* and the boundary edges of the conflict zone are output via the corresponding output iterators. The function returns in a std::pair the resulting output iterators.
Precondition: dimension()==2

template <class OutputItFaces, class OutputItHiddenVertices>
std::pair<OutputItFaces,OutputItHiddenVertices>

rt.get_conflicts_and_hidden_vertices(Weighted_point p,
OutputItFaces fit,
OutputItHiddenVertices vit,
Face_handle start) const

same as above except that only the faces in conflict with *p* and the vertices that would be hidden by *p* are output via the corresponding output iterators. The function returns in a std::pair the resulting output iterators.
Precondition: dimension()==2

template <class OutputItBoundaryEdges, class OutputItHiddenVertices>
std::pair<OutputItBoundaryEdges,OutputItHiddenVertices>

rt.get_boundary_of_conflicts_and_hidden_vertices(Weighted_point p,
OutputItBoundaryEdges eit,
OutputItHiddenVertices vit,
Face_handle start) const

same as above except that only the vertices that would be hidden by *p* and the boundary of the zone in conflict with *p* are output via the corresponding output iterators. The boundary edges of the conflict zone are output in counterclockwise order and each edge is described through the incident face which is not in conflict with *p*. The function returns in a std::pair the resulting output iterators.

template <class OutputItFaces>
OutputItFaces

rt.get_conflicts(Point p, OutputItFaces fit, Face_handle start) const

same as above except that only the faces in conflict with *p* are output. The function returns the resulting output iterator.
Precondition: dimension()==2

template <class OutputItBoundaryEdges>

OutputItBoundaryEdges

rt.get_boundary_of_conflicts(Point p, OutputItBoundaryEdges eit, Face_handle start) const

same as above except that only the boundary edges of the conflict zone are output in counterclockwise order where each edge is described through the incident face which is not in conflict with p. The function returns the resulting output iterator.

template <class OutputItHiddenVertices>
OutputItHiddenVertices

rt.get_hidden_vertices(Point p, OutputItHiddenVertices vit, Face_handle start) const

same as above except that only the vertices that would be hidden by p are output. The function returns the resulting output iterator.

Vertex_handle *rt.nearest_power_vertex(Bare_point p)*

Returns the vertex of the triangulation which is nearest to p with respect to the power distance. This means that the power of the query point p with respect to the weighted point in the nearest vertex is smaller than the power of p with respect to the weighted point in any other vertex. Ties are broken arbitrarily. The default constructed handle is returned if the triangulation is empty.

Access functions

int *rt.number_of_vertices() const*

returns the number of finite vertices that are not hidden.

int *rt.number_of_hidden_vertices() const*

returns the number of hidden vertices.

Hidden_vertices_iterator *rt.hidden_vertices_begin() const*

starts at an arbitrary hidden vertex.

Hidden_vertices_iterator *rt.hidden_vertices_end() const*

past the end iterator for the sequence of hidden vertices.

Finite_vertices_iterator *rt.finite_vertices_begin() const*

starts at an arbitrary unhidden finite vertex

Finite_vertices_iterator *rt.finite_vertices_end() const*

Past-the-end iterator

All_vertices_iterator *rt.all_vertices_end() const*

starts at an arbitrary unhidden vertex.

All_vertices_iterator *rt.all_vertices_begin() const*

past the end iterator.

Dual power diagram

The following member functions provide the elements of the dual power diagram.

Point *rt.weighted_circumcenter(Face_handle f) const*

returns the center of the circle orthogonal to the three weighted points corresponding to the vertices of face f.
Precondition: f is not infinite

Point *rt.dual(Face_handle f) const*

same as weighted_circumcenter

Object *rt.dual(Edge e) const* If both incident faces are finite, returns a segment whose endpoints are the duals of each incident face. If only one incident face is finite, returns a ray whose endpoint is the dual of the finite incident face and supported by the line which is the bisector of the edge's endpoints. If both incident faces are infinite, returns the line which is the bisector of the edge's endpoints otherwise.

Object *rt.dual(Edge_circulator ec) const*

Idem

Object *rt.dual(Edge_iterator ei) const*

Idem

template < class Stream>
Stream& *rt.draw_dual(Stream & ps)*

output the dual power diagram to stream *ps*.

Predicates

Oriented_side *rt.power_test(Face_handle f, Weighted_point p) const*

Returns the power test of p with respect to the power circle associated with f

advanced

Miscellaneous

bool *rt.is_valid(bool verbose = false, int level = 0) const*

Tests the validity of the triangulation as a *Triangulation_2* and additionally test the regularity of the triangulation. This method is useful to debug regular triangulation algorithms implemented by the user.

advanced

See Also

CGAL::Triangulation_2<Traits,Tds>,
TriangulationDataStructure_2,
RegularTriangulationTraits_2
RegularTriangulationFaceBase_2
RegularTriangulationVertexBase_2
CGAL::Regular_triangulation_face_base_2<Traits>
CGAL::Regular_triangulation_vertex_base_2<Traits>

CGAL::Regular_triangulation_euclidean_traits_2<K,Weight>

Class

Definition

Regular_triangulation_euclidean_traits_2<K,Weight> is a model for the concept *RegularTriangulationTraits_2* This traits class is templated by a kernel class *K* and a weight type *Weight*. This class inherits from *K* and uses a *Weighted_point* type derived from the type *K::Point_2*.

Note that this template class is specialized for *CGAL::Exact_predicates_inexact_constructions_kernel*, so that it is as if *Regular_triangulation_filtered_traits_2* was used, i.e. you get filtered predicates automatically.

#include <CGAL/Regular_triangulation_euclidean_traits_2.h>

Is Model for the Concepts

RegularTriangulationTraits_2

Inherits From

K

See Also

RegularTriangulationTraits_2
CGAL::Regular_triangulation_filtered_traits_2
CGAL::Regular_triangulation_2

Class

CGAL::Regular_triangulation_filtered_traits_2<FK>

Definition

The class *Regular_triangulation_filtered_traits_2<FK>* is designed as a traits class for the class *Regular_triangulation_2<RegularTriangulationTraits_2,TriangulationDataStructure_2>*. Its difference with *Regular_triangulation_euclidean_traits_2* is that it provides filtered predicates which are meant to be fast and exact.

The first argument *FK* must be a model of the *Kernel* concept, and it is also restricted to be an instance of the *Filtered_kernel* template.

#include <CGAL/Regular_triangulation_filtered_traits_2.h>

Is Model for the Concepts

RegularTriangulationTraits_2

Inherits From

Regular_triangulation_euclidean_traits_2<FK>

See Also

CGAL::Regular_triangulation_euclidean_traits_2.

CGAL::Regular_triangulation_face_base_2<Traits,Fb>

Definition

The class *Regular_triangulation_face_base_2<Traits,Fb>* is a model of the concept *RegularTriangulationFaceBase_2*. It is the default face base class of regular triangulations.

#include <CGAL/Regular_triangulation_face_base_2.h>

Parameters

The template parameters *Traits* has to be a model of *RegularTriangulationTraits_2*.

The template parameter *Fb* has to be a model of *TriangulationFaceBase_2*. By default, this parameter is instantiated by *CGAL::Triangulation_face_base_2<Traits>*.

Is Model for the Concepts

RegularTriangulationFaceBase_2

Inherits From

Fb

See Also

RegularTriangulationFaceBase_2
RegularTriangulationTraits_2
CGAL::Regular_triangulation_2<Traits,Tds>
CGAL::Regular_triangulation_vertex_base_2<Traits,Vb>

Class

CGAL::Regular_triangulation_vertex_base_2<Traits,Vb>

Definition

The class *Regular_triangulation_vertex_base_2<Traits,Vb>* is a model of the concept *RegularTriangulationVertexBase_2*. It is the default vertex base class of regular triangulations.

#include <CGAL/Regular_triangulation_vertex_base_2.h>

Parameters

The template parameters *Traits* has to be a model of *RegularTriangulationTraits_2*.

The template parameters *Vb* has to be a model of the concept *TriangulationVertexBase_2* and is by default instantiated by *CGAL::Triangulation_vertex_base_2<Traits>*.

Is Model for the Concepts

RegularTriangulationVertexBase_2

Inherits From

Vb

See Also

CGA::Triangulation_vertex_base_2<Traits,Vb>
CGAL::Regular_triangulation_2<Traits,Tds>
CGAL::Regular_triangulation_face_base_2<Traits,Fb>

TriangulationFaceBase_2

Concept

Definition

The concept TriangulationFaceBase_2 describes the requirements for the base face class of a triangulation data structure that is itself plugged into a basic triangulation or a Delaunay triangulation.

This concept refines the concept *TriangulationDSFaceBase_2* and could add geometric information. In fact, currently the triangulations of CGAL do not store any geometric information in the faces and, thus this concept is just equal to *TriangulationDSFaceBase_2* and only provided for symmetry with the vertex case.

Refines

TriangulationDSFaceBase_2

Has Models

CGAL::Triangulation_face_base_2<Traits>

See Also

TriangulationVertexBase_2
CGAL::Triangulation_face_base_2<Traits>
CGAL::Triangulation_2<Traits,Tds>
CGAL::Delaunay_triangulation_2<Traits,Tds>

Concept

TriangulationHierarchyVertexBase_2

Definition

The vertex of a triangulation included in a triangulation hierarchy has to provide some pointers to the corresponding vertices in the triangulations of the next and preceding levels. Therefore, the concept *TriangulationHierarchyVertexBase_2* refines the concept *TriangulationVertexBase_2*, adding handles to the corresponding vertices in the next and previous level triangulations.

Refines

TriangulationVertexBase_2

Operations

Vertex_handle	*v.up()*	returns the corresponding vertex (if any) of the next level triangulation;
Vertex_handle	*v.down()*	returns the corresponding vertex of the previous level triangulation;
void	*v.set_up(Vertex_handle u)*	sets the handle pointing to to the corresponding vertex of the next level triangulation;
void	*v.set_down(Vertex_handle d)*	sets the handle pointing to the corresponding vertex of the previous level triangulation;

Has Models

CGAL::Triangulation_hierarchy_vertex_base_2<Vb>

See Also

Triangulation_hierarchy_2<Tr>

TriangulationTraits_2

Definition

The concept TriangulationTraits_2 describes the set of requirements to be fulfilled by any class used to instantiate the first template parameter of the class *Triangulation_2<Traits,Tds>*. This concept provides the types of the geometric primitives used in the triangulation and some function object types for the required predicates on those primitives.

Types

TriangulationTraits_2:: Point_2	The point type.
TriangulationTraits_2:: Segment_2	The segment type.
TriangulationTraits_2:: Triangle_2	The triangle type.
TriangulationTraits_2:: Construct_segment_2	A constructor object for *Segment_2*. Provides : *Segment_2 operator()(Point_2 p,Point_2 q)*, which constructs a segment from two points.
TriangulationTraits_2:: Construct_triangle_2	A constructor object for *Triangle_2*. Provides : *Triangle_2 operator()(Point_2 p,Point_2 q,Point_2 r)*, which constructs a triangle from three points.
TriangulationTraits_2:: Less_x_2	Predicate object. Provides the operator : *bool operator()(Point p, Point q)* which returns *true* if p is before q according to the x-ordering of points.
TriangulationTraits_2:: Less_y_2	Predicate object. Provides the operator : *bool operator()(Point p, Point q)* which returns *true* if p is before q according to the y-ordering of points.
TriangulationTraits_2:: Compare_x_2	Predicate object. Provides the operator : *Comparison_result operator()(Point p, Point q)* which returns *SMALLER, EQUAL* or *LARGER* according to the x-ordering of points p and q.
TriangulationTraits_2:: Compare_y_2	Predicate object. Provides the operator : *Comparison_result operator()(Point p, Point q)* which returns (*SMALLER, EQUAL* or *LARGER*) according to the y-ordering of points p and q.
TriangulationTraits_2:: Orientation_2	Predicate object. Provides the operator : *Orientation operator()(Point p, Point q, Point r)* which returns *LEFT_TURN*, *RIGHT_TURN* or *COLLINEAR* depending on r being, with respect to the oriented line pq, on the left side , on the right side or on the line.

TriangulationTraits_2:: Side_of_oriented_circle_2

Predicate object. Must provide the operator *Oriented_side operator()(Point p, Point q, Point r, Point s)* which takes four points p,q,r,s as arguments and returns *ON_POSITIVE_SIDE*, *ON_NEGATIVE_SIDE* or, *ON_ORIENTED_BOUNDARY* according to the position of points s with respect to the oriented circle through through p,q and r. This type is required only if the function *side_of_oriented_circle(Face_handle f, Point p)* is called.

TriangulationTraits_2:: Construct_circumcenter_2

Constructor object. Provides the operator :
Point operator()(Point p, Point q, Point r)
which returns the circumcenter of the three points *p, q* and r. This type is required only if the function *Point circumcenter(Face_handle f)*is called.

Creation

Only a default constructor, copy constructor and an assignment operator are required. Note that further constructors can be provided.

TriangulationTraits_2 traits; — default constructor.

TriangulationTraits_2 traits(gtr); — Copy constructor

TriangulationTraits_2 $traits = gtr$ — Assignment operator.

Predicate functions

The following functions give access to the predicate and constructor objects.

Construct_segment_2	*traits.construct_segment_2_object()*
Construct_triangle_2	*traits.construct_triangle_2_object()*
Comparison_x_2	*traits.compare_x_2_object()*
Comparison_y_2	*traits.compare_y_2_object()*
Orientation_2	*traits.orientation_2_object()*
Side_of_oriented_circle_2	*traits.side_of_oriented_circle_2_object()*

Required only if *side_of_oriented_circle* is called called.

Construct_circumcenter_2

traits.construct_circumcenter_2_object()

Required only if *circumcenter* is called.

Has Models

All the CGAL Kernels

CGAL::Triangulation_euclidean_traits_2<K> . . . . page 2565

CGAL::Projection_traits_xy_3<K> . . . . page 604
CGAL::Projection_traits_yz_3<K>
CGAL::Projection_traits_zx_3<K>

deprecated

CGAL::Triangulation_euclidean_traits_xy_3<K> . . . . page 2566,
CGAL::Triangulation_euclidean_traits_yz_3<K>,
CGAL::Triangulation_euclidean_traits_zx_3<K>.

deprecated

See Also

CGAL::Triangulation_2<Traits,Tds>

Concept

TriangulationVertexBase_2

Definition

The concept TriangulationVertexBase_2 describes the requirements for the vertex base class of a triangulation data structure to be plugged in a basic, Delaunay or constrained triangulations.

The concept TriangulationVertexBase_2 refines the concept *TriangulationDSVertexBase_2* adding geometric information : the vertex base of a triangulation stores a point.

Refines

TriangulationDSVertexBase_2

Types

TriangulationVertexBase_2:: Point	Must be the same as the point type *TriangulationTraits_2::Point_2* defined by the geometric traits class of the triangulation.

Creation

TriangulationVertexBase_2 v(Point p); constructs a vertex embedded in point p.

TriangulationVertexBase_2 v(Point p, Face_handle f);

constructs a vertex embedded in point p and pointing on face f.

Access Functions

Point *v.point() const* returns the point.

Setting

void *v.set_point(Point p)*

sets the point.

I/O

istream& *istream& is >> & v*

Inputs the non-combinatorial information given by the vertex: the point and other possible information.

ostream&

ostream& os << *v*

Outputs the non combinatorial operation given by the vertex: the point and other possible information.

Has Models

CGAL::TriangulationVertexBase_2<Traits>.

See Also

TriangulationDataStructure_2
TriangulationDataStructure_2::Vertex
CGAL::Triangulation_vertex_base_2<Traits>

Concept

TriangulationVertexBaseWithInfo_2

Definition

A type model of this concept can be used as vertex base by a triangulation and provides an additional information storage.

Refines

TriangulationVertexBase_2

Types

TriangulationVertexBaseWithInfo_2:: Info	A type which is *DefaultConstructible* and *Assignable*.

Access Functions

Info	*v.info() const*	Returns a const reference to the object of type *Info* stored in the vertex.
Info&	*v.info()*	Returns a reference to the object of type *Info* stored in the vertex.

Has Models

CGAL::Triangulation_vertex_base_with_info_2

CGAL::Triangulation_2<Traits,Tds>

Definition

The class *Triangulation_2<Traits,Tds>* is the basic class designed to handle triangulations of set of points A in the plane.

Such a triangulation has vertices at the points of A and its domain covers the convex hull of A. It can be viewed as a planar partition of the plane whose bounded faces are triangular and cover the convex hull of A. The single unbounded face of this partition is the complementary of the convex hull of A.

In many applications, it is convenient to deal only with triangular faces. Therefore, we add to the triangulation a fictitious vertex, called the *infinite vertex* and we make each convex hull edge incident to an *infinite* face having as third vertex the *infinite vertex*. In that way, each edge is incident to exactly two faces and special cases at the boundary of the convex hull are simpler to deal with.

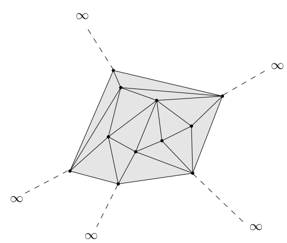


Figure 37.10: The infinite vertex.

The class *Triangulation_2<Traits,Tds>* implements this point of view and therefore considers the triangulation of the set of points as a set of triangular, finite and infinite faces. Although it is convenient to draw a triangulation as in figure 37.10, note that the *infinite vertex* has no significant coordinates and that no geometric predicate can be applied on it or on an infinite face.

A triangulation is a collection of vertices and faces that are linked together through incidence and adjacency relations. Each face give access to its three incident vertices and to its three adjacent faces. Each vertex give access to one of its incident faces.

The three vertices of a face are indexed with 0, 1 and 2 in counterclockwise order. The neighbor of a face are also indexed with 0,1,2 in such a way that the neighbor indexed by i is opposite to the vertex with the same index.

The triangulation class offer two functions *int cw(int i)* and *int ccw(int i)* which given the index of a vertex in a face compute the index of the next vertex of the same face in clockwise or counterclockwise order. Thus, for example the neighbor *neighbor(cw(i))* is the neighbor of f which is next to *neighbor(i)* turning clockwise around f. The face *neighbor(cw(i))* is also the first face encountered after f when turning clockwise around vertex i of f (see Figure 37.11).

#include <CGAL/Triangulation_2.h>

Parameters

The class *Triangulation_2<Traits,Tds>* has two template parameters. The first one *Traits* is the geometric traits, it is to be instantiated by a model of the concept *TriangulationTraits_2*.

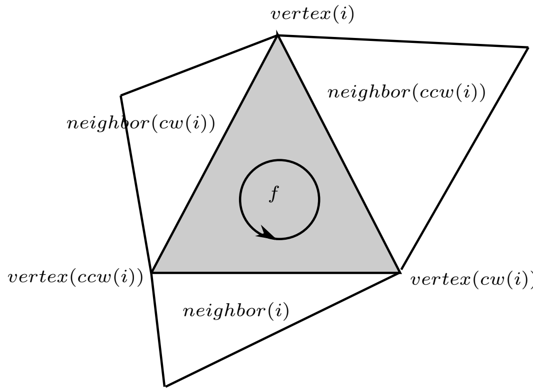


Figure 37.11: Vertices and neighbors.

The second parameter is the triangulation data structure, it has to be instantiated by a model of the concept *TriangulationDataStructure_2*. By default, the triangulation data structure is instantiated by *CGAL::Triangulation_data_structure_2 < CGAL::Triangulation_vertex_base_2<Gt>, CGAL::Triangulation_face_base_2<Gt> >*.

Inherits From

Triangulation_cw_ccw_2 This class provides the functions *cw(i)* and *ccw(i)*.

Types

typedef Traits	*Geom_traits;*	the traits class.
typedef Tds	*Triangulation_data_structure;*	the triangulation data structure type.
typedef Traits::Point_2	*Point;*	the point type
typedef Traits::Segment_2	*Segment;*	the segment type
typedef Traits::Triangle_2	*Triangle;*	the triangle type
typedef Tds::Vertex	*Vertex;*	the vertex type.
typedef Tds::Face	*Face;*	the face type.
typedef Tds::Edge	*Edge;*	the edge type.
typedef Tds::size_type	*size_type;*	Size type (an unsigned integral type)
typedef Tds::difference_type	*difference_type;*	Difference type (a signed integral type)

The vertices and faces of the triangulations are accessed through *handles*, *iterators* and *circulators*. The handles are models of the concept *Handle* which basically offers the two dereference operators * and ->. The iterators and circulators are all bidirectional and non mutable. The circulators and iterators are convertible to handles with the same value type, so that whenever a handle appear in the parameter list of a function, an appropriate iterator or circulator can be passed as well.

The edges of the triangulation can also be visited through iterators and circulators, the edge circulators and iterators are also bidirectional and non mutable.

In the following, we called *infinite* any face or edge incident to the infinite vertex and the infinite vertex itself. Any other feature (face, edge or vertex) of the triangulation is said to be *finite*. Some iterators (the *All* iterators

) allows to visit finite or infinite feature while others (the *Finite* iterators) visit only finite features. Circulators visit infinite features as well as finite ones.

typedef Tds::Vertex_handle	*Vertex_handle;*	handle to a vertex
typedef Tds::Face_handle	*Face_handle;*	handle to a face

typedef Tds::Face_iterator	*All_faces_iterator;*	iterator over all faces.
typedef Tds::Edge_iterator	*All_edges_iterator;*	iterator over all edges
typedef Tds::Vertex_iterator	*All_vertices_iterator;*	iterator over all vertices

Triangulation_2<Traits,Tds>:: Finite_faces_iterator	iterator over finite faces.
Triangulation_2<Traits,Tds>:: Finite_edges_iterator	iterator over finite edges.
Triangulation_2<Traits,Tds>:: Finite_vertices_iterator	iterator over finite vertices.
Triangulation_2<Traits,Tds>:: Point_iterator	iterator over the points corresponding the finite vertices of the triangulation.

Triangulation_2<Traits,Tds>:: Line_face_circulator	circulator over all faces intersected by a line.

Triangulation_2<Traits,Tds>:: Face_circulator	circulator over all faces incident to a given vertex.
Triangulation_2<Traits,Tds>:: Edge_circulator	circulator over all edges incident to a given vertex.
Triangulation_2<Traits,Tds>:: Vertex_circulator	circulator over all vertices incident to a given vertex.

The triangulation class also defines the following enum type to specify which case occurs when locating a point in the triangulation.

enum Locate_type { *VERTEX=0, EDGE, FACE, OUTSIDE_CONVEX_HULL, OUTSIDE_AFFINE_HULL*}*;*

The locate type is *OUTSIDE_CONVEX_HULL* when the point is outside the convex hull but in the affine hull of the current triangulation.
The locate type is *OUTSIDE_AFFINE_HULL* when the point is outside the affine hull of the current triangulation.

Creation

Triangulation_2<Traits,Tds> t; default constructor.

Triangulation_2<Traits,Tds> t(Traits gt = Traits());

Introduces an empty triangulation *t*.

Triangulation_2<Traits,Tds> t(Triangulation_2 tr);

Copy constructor. All the vertices and faces are duplicated. After the copy, *t* and *tr* refer to different triangulations : if *tr* is modified, *t* is not.

Triangulation_2	$t = tr$	Assignment. All the vertices and faces are duplicated. After the assignment, *t* and *tr* refer to different triangulations : if *tr* is modified, *t* is not.
void	*t.swap(Triangulation_2& tr)*	
		The triangulations *tr* and *t* are swapped. *t.swap(tr)* should be preferred to *t = tr* or to *t(tr)* if *tr* is deleted after that.
void	*t.clear()*	Deletes all faces and finite vertices resulting in an empty triangulation.

Access Functions

Geom_traits	*t.geom_traits() const*	Returns a const reference to the triangulation traits object.
TriangulationDataStructure_2		
	t.tds() const	Returns a const reference to the triangulation data structure.

advanced

Non const access

The responsibility of keeping a valid triangulation belongs to the user when using advanced operations allowing a direct manipulation of the *tds*.

TriangulationDataStructure_2&

	t.tds()	Returns a reference to the triangulation data structure.

This method is mainly a help for users implementing their own triangulation algorithms.

advanced

int	*t.dimension() const*	Returns the dimension of the convex hull.
size_type	*t.number_of_vertices() const*	
		Returns the number of finite vertices.
size_type	*t.number_of_faces() const*	
		Returns the number of finite faces.
Face_handle	*t.infinite_face() const*	a face incident to the *infinite_vertex*.
Vertex_handle	*t.infinite_vertex()*	the *infinite_vertex*.
Vertex_handle	*t.finite_vertex() const*	a vertex distinct from the *infinite_vertex*.

Predicates

The class *Triangulation_2<Traits,Tds>* provides methods to test the finite or infinite character of any feature, and also methods to test the presence in the triangulation of a particular feature (edge or face).

bool *t.is_infinite(Vertex_handle v) const*

true iff *v* is the *infinite_vertex*.

bool *t.is_infinite(Face_handle f) const* *true* iff face *f* is infinite.

bool *t.is_infinite(Face_handle f, int i) const*

true iff edge *(f,i)* is infinite.

bool *t.is_infinite(Edge e) const* *true* iff edge *e* is infinite.

bool *t.is_infinite(Edge_circulator ec) const*

true iff edge **ec* is infinite.

bool *t.is_infinite(Edge_iterator ei) const*

true iff edge **ei* is infinite.

bool *t.is_edge(Vertex_handle va, Vertex_handle vb)*

true if there is an edge having *va* and *vb* as vertices.

bool *t.is_edge(Vertex_handle va, Vertex_handle vb, Face_handle& fr, int & i)*

as above. In addition, if *true* is returned, the edge with vertices *va* and *vb* is the edge *e=(fr,i)* where *fr* is a handle to the face incident to *e* and on the right side of *e* oriented from *va* to *vb*.

bool *t.includes_edge(Vertex_handle va, Vertex_handle & vb, Face_handle& fr, int & i)*

true if the line segment from *va* to *vb* includes an edge *e* incident to *va*. If *true*, *vb* becomes the other vertex of *e*, *e* is the edge *(fr,i)* where *fr* is a handle to the face incident to *e* and on the right side *e* oriented from *va* to *vb*.

bool *t.is_face(Vertex_handle v1, Vertex_handle v2, Vertex_handle v3)*

true if there is a face having *v1*, *v2* and *v3* as vertices.

bool *t.is_face(Vertex_handle v1, Vertex_handle v2, Vertex_handle v3, Face_handle &fr)*

as above. In addition, if *true* is returned, fr is a handle to the face with *v1*, *v2* and *v3* as vertices.

Queries

The class *Triangulation_2<Traits,Tds>* provides methods to locate a given point with respect to a triangulation. It also provides methods to locate a point with respect to a given finite face of the triangulation.

Face_handle *t.locate(Point query, Face_handle f = Face_handle())*

If the point *query* lies inside the convex hull of the points, a face that contains the query in its interior or on its boundary is returned.
If the point *query* lies outside the convex hull of the triangulation but in the affine hull, the returned face is an infinite face which is a proof of the point's location :
- for a two dimensional triangulation, it is a face (∞, p, q) such that *query* lies to the left of the oriented line *pq* (the rest of the triangulation lying to the right of this line).
- for a degenerate one dimensional triangulation it is the (degenerate one dimensional) face $(\infty, p, NULL)$ such that *query* and the triangulation lie on either side of *p*.
If the point *query* lies outside the affine hull, the returned *Face_handle* is *NULL*.
The optional *Face_handle* argument, if provided, is used as a hint of where the locate process has to start its search.

Face_handle *t.locate(Point query, Locate_type& lt, int& li, Face_handle h =Face_handle())*

Same as above. Additionally, the parameters *lt* and *li* describe where the query point is located. The variable *lt* is set to the locate type of the query. If *lt==VERTEX* the variable *li* is set to the index of the vertex, and if *lt==EDGE li* is set to the index of the vertex opposite to the edge. Be careful that *li* has no meaning when the query type is *FACE*, *OUTSIDE_CONVEX_HULL*, or *OUTSIDE_AFFINE_HULL* or when the triangulation is 0-dimensional.

Oriented_side *t.oriented_side(Face_handle f, Point p) const*

Returns on which side of the oriented boundary of f lies the point p.
Precondition: f is finite.

Oriented_side *t.side_of_oriented_circle(Face_handle f, Point p)*

Returns on which side of the circumcircle of face f lies the point p. The circle is assumed to be counterclockwise oriented, so its positive side correspond to its bounded side. This predicate is available only if the corresponding predicates on points is provided in the geometric traits class.

Modifiers

The following operations are guaranteed to lead to a valid triangulation when they are applied on a valid triangulation.

void *t.flip(Face_handle f, int i)*

Exchanges the edge incident to f and *f->neighbor(i)* with the other diagonal of the quadrilateral formed by f and *f->neighbor(i)*.
Precondition: The faces f and *f->neighbor(i)* are finite faces and their union form a convex quadrilateral.

Vertex_handle *t.insert(Point p, Face_handle f = Face_handle())*

Inserts point p in the triangulation and returns the corresponding vertex.
If point p coincides with an already existing vertex, this vertex is returned and the triangulation remains unchanged.
If point p is on an edge, the two incident faces are split in two.
If point p is strictly inside a face of the triangulation, the face is split in three.
If point p is strictly outside the convex hull, p is linked to all visible points on the convex hull to form the new triangulation.
At last, if p is outside the affine hull (in case of degenerate 1-dimensional or 0-dimensional triangulations), p is linked all the other vertices to form a triangulation whose dimension is increased by one. The last argument f is an indication to the underlying locate algorithm of where to start.

Vertex_handle *t.insert(Point p, Locate_type lt, Face_handle loc, int li)*

Same as above except that the location of the point p to be inserted is assumed to be given by *(lt,loc,i)* (see the description of the *locate* method above.)

Vertex_handle *t.push_back(Point p)*

Equivalent to *insert(p)*.

template < class InputIterator >
std::ptrdiff_t *t.insert(InputIterator first, InputIterator last)*

Inserts the points in the range $[first, last)$. Returns the number of inserted points.
Precondition: The *value_type* of *InputIterator* is *Point*.

void *t.remove(Vertex_handle v)*

Removes the vertex from the triangulation. The created hole is re-triangulated.
Precondition: Vertex v must be finite.

Vertex_handle *t.move_if_no_collision(Vertex_handle v, Point p)*

if there is not already another vertex placed on p, the triangulation is modified such that the new position of vertex v is p, and v is returned. Otherwise, the triangulation is not modified and the vertex at point p is returned.
Precondition: Vertex v must be finite.

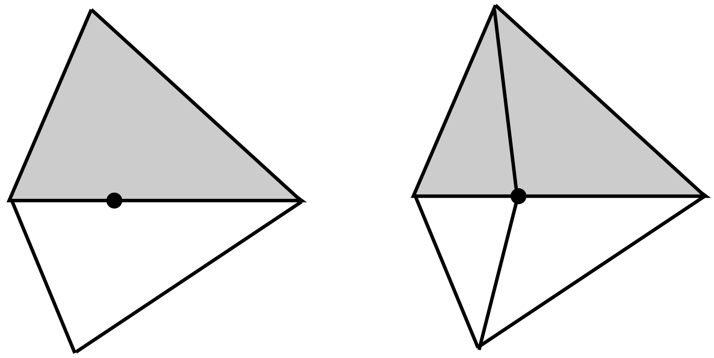


Figure 37.12: Insertion of a point on an edge.

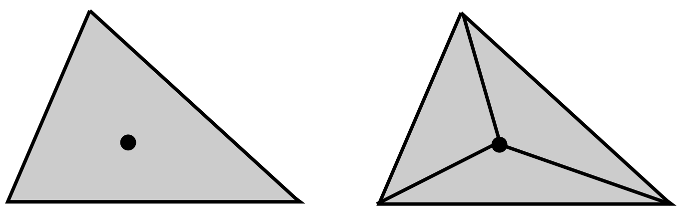


Figure 37.13: Insertion in a face.

Vertex_handle *t.move(Vertex_handle v, Point p)*

same as above if there is no collision. Otherwise, v is deleted and the vertex placed on p is returned.
Precondition: Vertex v must be finite.

advanced

The following member functions offer more specialized versions of the insertion or removal operations to be used when one knows to be in the corresponding case.

Vertex_handle *t.insert_first(Point p)*

Inserts the first finite vertex .

Vertex_handle *t.insert_second(Point p)*

Inserts the second finite vertex .

Vertex_handle *t.insert_in_face(Point p, Face_handle f)*

Inserts vertex v in face f. Face f is modified, two new faces are created.
Precondition: The point in vertex v lies inside face f.

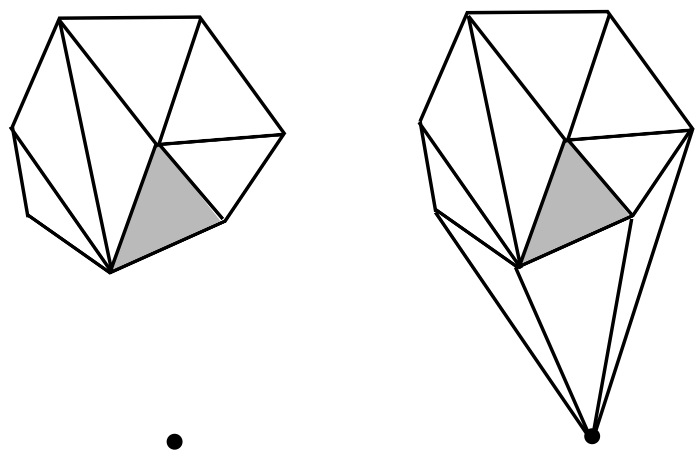


Figure 37.14: Insertion outside the convex hull.

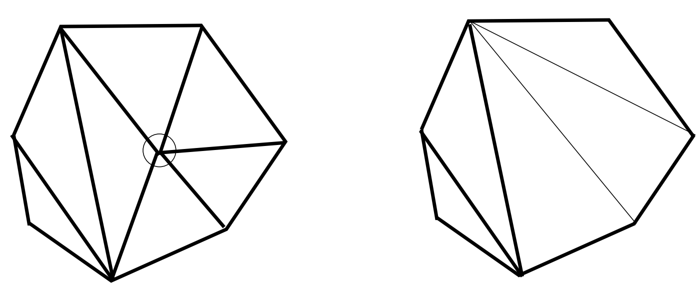


Figure 37.15: Removal

Vertex_handle *t.insert_in_edge(Point p, Face_handle f, int i)*

Inserts vertex v in edge *i* of *f*.
Precondition: The point in vertex *v* lies on the edge opposite to the vertex *i* of face *f*.

Vertex_handle *t.insert_outside_convex_hull(Point p, Face_handle f)*

Inserts a point which is outside the convex hull but in the affine hull.
Precondition: The handle *f* points to a face which is a proof of the location of*p*, see the description of the *locate* method above.

Vertex_handle *t.insert_outside_affine_hull(Point p)*

Inserts a point which is outside the affine hull.

void *t.remove_degree_3(Vertex_handle v)*

Removes a vertex of degree three. Two of the incident faces are destroyed, the third one is modified.
Precondition: Vertex *v* is a finite vertex with degree three.

void *t.remove_second(Vertex_handle v)*

Removes the before last finite vertex.

void *t.remove_first(Vertex_handle v)*

Removes the last finite vertex.

The following functions are mainly intended to be used in conjunction with the *find_conflicts()* member functions of Delaunay and constrained Delaunay triangulations to perform insertions.

template<class EdgeIt>
Vertex_handle *t.star_hole(Point p, EdgeIt edge_begin, EdgeIt edge_end)*

creates a new vertex *v* and use it to star the hole whose boundary is described by the sequence of edges *[edge_begin, edge_end[*. Returns a handle to the new vertex.

template<class EdgeIt, class FaceIt>
Vertex_handle *t.star_hole(Point p,*
EdgeIt edge_begin,
EdgeIt edge_end,
FaceIt face_begin,
FaceIt face_end)

same as above, except that the algorithm first recycles faces in the sequence *[face_begin, face_end[* and create new ones only when the sequence is exhausted.

advanced

Traversal of the Triangulation

A triangulation can be seen as a container of faces and vertices. Therefore the triangulation provides several iterators and circulators that allow to traverse it (completely or partially).

Face, Edge and Vertex Iterators

The following iterators allow respectively to visit finite faces, finite edges and finite vertices of the triangulation. These iterators are non mutable, bidirectional and their value types are respectively *Face*, *Edge* and *Vertex*. They are all invalidated by any change in the triangulation.

Finite_vertices_iterator *t.finite_vertices_begin() const*

Starts at an arbitrary finite vertex

Finite_vertices_iterator *t.finite_vertices_end() const*

Past-the-end iterator

Finite_edges_iterator *t.finite_edges_begin() const*

Starts at an arbitrary finite edge

Finite_edges_iterator *t.finite_edges_end() const*

Past-the-end iterator

Finite_faces_iterator *t.finite_faces_begin() const*

Starts at an arbitrary finite face

Finite_faces_iterator *t.finite_faces_end() const*

Past-the-end iterator

Point_iterator *t.points_begin() const*

Point_iterator *t.points_end() const* Past-the-end iterator

The following iterators allow respectively to visit all (finite or infinite) faces, edges and vertices of the triangulation. These iterators are non mutable, bidirectional and their value types are respectively *Face*, *Edge* and *Vertex*. They are all invalidated by any change in the triangulation.

All_vertices_iterator *t.all_vertices_begin() const*

Starts at an arbitrary vertex

All_vertices_iterator *t.all_vertices_end() const*

Past-the-end iterator

All_edges_iterator	*t.all_edges_begin() const*	
		Starts at an arbitrary edge
All_edges_iterator	*t.all_edges_end() const*	Past-the-end iterator
All_faces_iterator	*t.all_faces_begin() const*	Starts at an arbitrary face
All_faces_iterator	*t.all_faces_end() const*	Past-the-end iterator

Line Face Circulator

The triangulation defines a circulator that allows to visit all faces that are intersected by a line. A face f is considered has being intersected by the oriented line l if either:

- f is a finite face whose interior intersects l, or
- f is a finite face with an edge collinear with l and lies to the left of l, or
- f is an infinite face incident to a convex hull edge whose interior is intersected by l, or
- f is an infinite face incident to a convex hull vertex lying on l and the finite edge of f lies to the left of l.

The circulator has a singular value if the line l intersect no finite face of the triangulation. This circulator is non-mutable and bidirectional. Its value type is *Face*.

Line_face_circulator *t.line_walk(Point p, Point q, Face_handle f = Face_handle())*

This function returns a circulator that allows to visit the faces intersected by the line pq. If there is no such face the circulator has a singular value.
The starting point of the circulator is the face f, or the first finite face traversed by l , if f is omitted.
The circulator wraps around the *infinite_vertex* : after the last traversed finite face, it steps through the infinite face adjacent to this face then through the infinite face adjacent to the first traversed finite face then through the first finite traversed face again.
Precondition: The triangulation must have dimension 2.
Precondition: Points p and q must be different points.
Precondition: If *f != NULL*, it must point to a finite face and the point p must be inside or on the boundary of f.

Figure 37.16 illustrates which finite faces are enumerated. Lines l_1 and l_2 have no face to their left. Lines l_3 and l_4 have faces to their left. Note that the finite faces that are only vertex incident to lines l_3 and l_4 are not enumerated.

A line face circulator is invalidated if the face the circulator refers to is changed.

Face, Edge and Vertex Circulators

The triangulation also provides circulators that allows to visit respectively all faces or edges incident to a given vertex or all vertices adjacent to a given vertex. These circulators are non-mutable and bidirectional. The *operator++* moves the circulator counterclockwise around the vertex while the *operator--* moves clockwise.

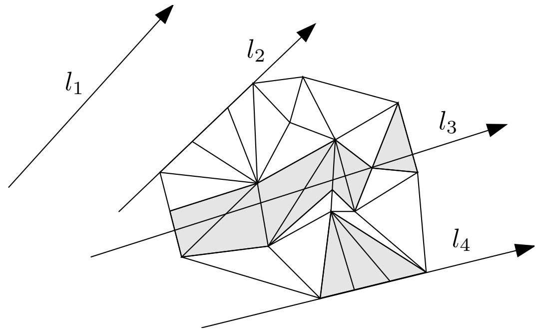


Figure 37.16: The line face circulator.

A face circulator is invalidated by any modification of the face pointed to. An edge or a vertex circulator are invalidated by any modification of one of the two faces incident to the edge pointed to.

Face_circulator *t.incident_faces(Vertex_handle v) const*

Starts at an arbitrary face incident to *v*.

Face_circulator *t.incident_faces(Vertex_handle v, Face_handle f) const*

Starts at face *f*.
Precondition: Face *f* is incident to vertex *v*.

Edge_circulator *t.incident_edges(Vertex_handle v) const*

Starts at an arbitrary edge incident to *v*.

Edge_circulator *t.incident_edges(Vertex_handle v, Face_handle f) const*

Starts at the first edge of *f* incident to *v*, in counterclockwise order around *v*.
Precondition: Face *f* is incident to vertex *v*.

Vertex_circulator *t.incident_vertices(Vertex_handle v) const*

Starts at an arbitrary vertex incident to *v*.

Vertex_circulator *t.incident_vertices(Vertex_handle v, Face_handle f)*

Starts at the first vertex of *f* adjacent to *v* in counterclockwise order around *v*.
Precondition: Face *f* is incident to vertex *v*.

Traversal of the Convex Hull

Applied on the *infinite_vertex* the above functions allow to visit the vertices on the convex hull and the infinite edges and faces. Note that a counterclockwise traversal of the vertices adjacent to the *infinite_vertex* is a clockwise traversal of the convex hull.

Face_circulator *t.incident_faces(t.infinite_vertex())*
Face_circulator *t.incident_faces(t.infinite_vertex(), Face_handle f) const*
Edge_circulator *t.incident_edges(t.infinite_vertex())*

Edge_circulator	*t.incident_edges(t.infinite_vertex(), Face_handle f)*
Vertex_circulator	*t.incident_vertices(t.infinite_vertex() v)*
Vertex_circulator	*t.incident_vertices(t.infinite_vertex(), Face_handle f)*

Traversal between adjacent faces

Vertex_handle *t.mirror_vertex(Face_handle f, int i) const*

returns the vertex of the i^{th} neighbor of f that is opposite to f.
Precondition: $0 \leq i \leq 2$.

int *t.mirror_index(Face_handle f, int i) const*

returns the index of f in its i^{th} neighbor.
Precondition: $0 \leq i \leq 2$.

Edge *t.mirror_edge(Edge e) const*

returns the same edge seen from the other adjacent face.
Precondition: $0 \leq i \leq 2$.

Miscellaneous

int *t.ccw(int i) const* Returns $i+1$ modulo 3.
Precondition: $0 \leq i \leq 2$.

int *t.cw(int i) const* Returns $i+2$ modulo 3.
Precondition: $0 \leq i \leq 2$.

Triangle

t.triangle(Face_handle f) const

Returns the triangle formed by the three vertices of f.
Precondition: The face is finite.

Segment *t.segment(Face_handle f, int i) const*

Returns the line segment formed by the vertices *ccw(i)* and *cw(i)* of face f.
Precondition: $0 \leq i \leq 2$. The vertices *ccw(i)* and *cw(i)* of f are finite.

Segment *t.segment(Edge e) const* Returns the line segment corresponding to edge e.
Precondition: e is a finite edge

Segment *t.segment(Edge_circulator ec) const*

Returns the line segment corresponding to edge **ec*.
Precondition: **ec* is a finite edge.

Segment *t.segment(Edge_iterator ei) const*

Returns the line segment corresponding to edge **ei*.
Precondition: **ei* is a finite edge.

Point *t.circumcenter(Face_handle f) const*

Compute the circumcenter of the face pointed to by f. This function is available only if the corresponding function is provided in the geometric traits.

advanced

Setting

void *t.set_infinite_vertex(Vertex_handle v)*

Checking

The responsibility of keeping a valid triangulation belongs to the users if advanced operations are used. Obviously the advanced user, who implements higher levels operations may have to make a triangulation invalid at some times. The following method is provided to help the debugging.

bool *t.is_valid(bool verbose = false, int level = 0) const*

Checks the combinatorial validity of the triangulation and also the validity of its geometric embedding. This method is mainly a debugging help for the users of advanced features.

advanced

I/O

The I/O operators are defined for *iostream*. The format for the iostream is an internal format.

ostream& *ostream& os << T*

Inserts the triangulation *t* into the stream *os*.
Precondition: The insert operator must be defined for *Point*.

istream& *istream& is >> T* Reads a triangulation from stream *is* and assigns it to *t*.
Precondition: The extract operator must be defined for *Point*.

The information output in the *iostream* is:
- the number of vertices (including the infinite one), the number of faces (including infinite ones), and the dimension.
- for each vertex (except the infinite vertex), the non combinatorial information stored in that vertex (point, etc.).
- for each faces, the indices of its vertices and the non combinatorial information (if any) in this face.
- for each face again the indices of the neighboring faces.
The index of an item (vertex of face) is the rank of this item in the output order. When dimension < 2, the same information is output for faces of maximal dimension instead of faces.

CGAL also provides a stream operator << to draw triangulations for *CGAL::Qt_widget*, the Qt based graphic package. These operators require the include statement :
#include CGAL/IO/Qt_widget_Triangulation_2.h
See the *Qt_widget* class.

Implementation

Locate is implemented by a line walk from a vertex of the face given as optional parameter (or from a finite vertex of *infinite_face()* if no optional parameter is given). It takes time $O(n)$ in the worst case, but only $O(\sqrt{n})$ on average if the vertices are distributed uniformly at random.

Insertion of a point is done by locating a face that contains the point, and then splitting this face. If the point falls outside the convex hull, the triangulation is restored by flips. Apart from the location, insertion takes a time time $O(1)$. This bound is only an amortized bound for points located outside the convex hull.

Removal of a vertex is done by removing all adjacent triangles, and re-triangulating the hole. Removal takes time $O(d^2)$ in the worst case, if d is the degree of the removed vertex, which is $O(1)$ for a random vertex.

The face, edge, and vertex iterators on finite features are derived from their counterparts visiting all (finite and infinite) features which are themselves derived from the corresponding iterators of the triangulation data structure.

See Also

TriangulationTraits_2
TriangulationDataStructure_2
TriangulationDataStructure_2::Face
TriangulationDataStructure_2::Vertex
CGAL::Triangulation_data_structure_2<Vb,Fb>
CGAL::Triangulation_vertex_base_2<Traits>
CGAL::Triangulation_face_base_2<Traits>

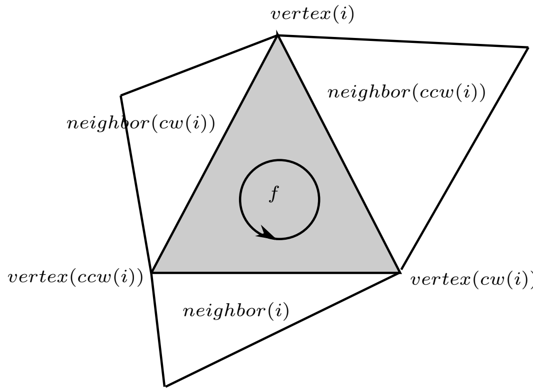


Figure 37.17: Vertices and neighbors.

CGAL::Triangulation_cw_ccw_2

Class

Definition

The class *Triangulation_cw_ccw_2* offer two functions *int cw(int i)* and *int ccw(int i)* which given the index of a vertex in a face compute the index of the next vertex of the same face in clockwise or counterclockwise order. This works also for neighbor indexes. Thus, for example the neighbor *neighbor(cw(i))* of a face f is the neighbor which is next to *neighbor(i)* turning clockwise around f. The face *neighbor(cw(i))* is also the first face encountered after f when turning clockwise around vertex i of f.

Many of the classes in the triangulation package inherit from *Triangulation_cw_ccw_2*. This is for instance the case for *CGAL::Triangulation_2<Traits,Tds>::Face*. Thus, for example the neighbor *neighbor(cw(i))* of a face f is the neighbor which is next to *neighbor(i)* turning clockwise around f. The face *neighbor(cw(i))* is also the first face encountered after f when turning clockwise around vertex i of f.

#include <CGAL/Triangulation_2.h>

Creation

Triangulation_cw_ccw_2 a; default constructor.

Operations

int *a.ccw(const int i) const*

returns the index of the neighbor or vertex that is next to the neighbor or vertex with index i in counterclockwise order around a face.

int *a.cw(const int i) const*

returns the index of the neighbor or vertex that is next to the neighbor or vertex with index i in counterclockwise order around a face.

See Also

CGAL::Triangulation_2<Traits,Tds>
CGAL::TriangulationDSFace_2

CGAL::Triangulation_euclidean_traits_2<K>

Class

Definition

The class *Triangulation_euclidean_traits_2<K>* can be used to instantiate the geometric traits class of basic and Delaunay triangulations. The templated parameter *K* has to be instantiated by a model of the *Kernel* concept. The class *Triangulation_euclidean_traits_2<K>* uses types and predicates defined *K*.

#include <CGAL/Triangulation_euclidean_traits_2.h>

Is Model for the Concepts

TriangulationTraits_2
DelaunayTriangulationTraits_2

See Also

TriangulationTraits_2
DelaunayTriangulationTraits_2
CGAL::Triangulation_2<Traits,Tds>
CGAL::Delaunay_triangulation_2<Traits,Tds>
CGAL::Triangulation_euclidean_traits_xy_3<K>

CGAL::Triangulation_euclidean_traits_xy_3<K>

The functionality of this class has been generalized to other packages than 2D triangulations. The more general class *Projection_traits_xy_3* can be found in the 2D and 3D Linear Geometric Kernel.

deprecated

Definition

The class *Triangulation_euclidean_traits_xy_3<K>* is a geometric traits class which allows to triangulate a terrain. This traits class is designed to build a two dimensional triangulation embedded in 3D space, i.e. a triangulated surface, such that its on the *xy* plane is a Delaunay triangulation. This is a usual construction for GIS terrains. Instead of really projecting the 3D points and maintaining a mapping between each point and its projection (which costs space and is error prone) the class *Triangulation_euclidean_traits_xy_3<K>* supplies geometric predicates that ignore the *z*-coordinate of the points.

The class is a model of the concept *DelaunayTriangulationTraits_2* except that it does not provide the type and constructors required to build the dual Voronoi diagram. The class is also a model of the concept *ConstrainedTriangulationTraits_2*.

Parameters

The template parameter *K* has to be instantiated by a model of the *Kernel* concept. *Triangulation_euclidean_traits_xy_3<K>* uses types and predicates defined in *K*.

#include <CGAL/Triangulation_euclidean_traits_xy_3.h>

Types

typedef Point_3<K> *Point_2;*
typedef Segment_3<K>
Segment_2;
typedef Triangle_3<K>
Triangle_2;
typedef Line_3<K> *Line_2;*

The following predicates and constructor types are provided

Triangulation_euclidean_traits_xy_3<K>:: Construct_segment_2

A constructor object for *Segment_2*. Provides :
Segment_2 operator()(Point_2 p,Point_2 q),
which constructs a segment from two points.

Triangulation_euclidean_traits_xy_3<K>:: Construct_triangle_2

A constructor object for *Triangle_2*. Provides :
Triangle_2 operator()(Point_2 p,Point_2 q,Point_2 r),
which constructs a triangle from three points.

Triangulation_euclidean_traits_xy_3<K>:: Construct_line_2

A constructor object for *Line_2*. Provides :
Line_2 operator()(Point_2 p,Point_2 q),
which constructs a line from two points.

Triangulation_euclidean_traits_xy_3<K>:: Compute_squared_distance_2

A construction object. Provides the operator :
RT operator()(Line_2 l, Point_2 p); which returns the squared distance between the projection of *p* and the projection of *l*.

Triangulation_euclidean_traits_xy_3<K>:: Intersect_2

A construction object. Provides the operator :
Object_2 operator()(Segment_2 s1, Segment_2 s2); which returns the intersection of the projection of *s1* and the projection of *s2* embedded in *3D*. If the intersection is a segment, the *z*-coordinates of its extremities is *0*. If the intersection is a point *p*, let *p1* and *p2* be the points on *s1* and *s2* respectively, such that their projections are *p*. The point returned is the middle of the segment *p1p2*.
Precondition: The projection of *s1* and the projection of *s2* are non-degenerate *2D* segments.

Triangulation_euclidean_traits_xy_3<K>:: Compare_x_2

Predicate object. Provides the operator :
Comparison_result operator()(Point_2 p, Point_2 q)
which returns *SMALLER, EQUAL* or *LARGER* according to the *x*-ordering of points *p* and *q*.

Triangulation_euclidean_traits_xy_3<K>:: Compare_y_2

Predicate object. Provides the operator :
Comparison_result operator()(Point_2 p, Point_2 q)
which returns (*SMALLER, EQUAL* or *LARGER*) according to the *y*-ordering of points *p* and *q*.

Triangulation_euclidean_traits_xy_3<K>:: Orientation_2

Predicate object. Provides the operator :
Orientation operator()(Point_2 p, Point_2 q, Point_2 r)
which returns *LEFT_TURN*, *RIGHT_TURN* or *COLLINEAR* according to the position of the projection of *r* with respect to the projection of the oriented line *pq*.

Triangulation_euclidean_traits_xy_3<K>:: Side_of_oriented_circle_2

Predicate object. Provides the operator : *Oriented_side operator()(Point_2 p, Point_2 q, Point_2 r, Point_2 s)* which takes four points p,q,r,s as arguments and returns *ON_POSITIVE_SIDE*, *ON_NEGATIVE_SIDE* or, *ON_ORIENTED_BOUNDARY* according to the position of the projection of point*s* with respect to the oriented circle through the projections of p,q and *r*.

Triangulation_euclidean_traits_xy_3<K>:: Compare_distance_2

Predicate object. Provides the operator :
Comparison_result operator()(Point_2 p, Point_2 q, Point_2 r) which returns *SMALLER*, *EQUAL* or *LARGER* according to the distance between the projection of p and the projection of q being smaller, equal or larger than the distance between the projection of p and the projection of r.

Creation

Triangulation_euclidean_traits_xy_3<K> traits;

default constructor.

Triangulation_euclidean_traits_xy_3<K> traits(Triangulation_euclidean_traits_xy_3 tr);

Copy constructor.

Triangulation_euclidean_traits_xy_3 *traits = Triangulation_euclidean_traits_xy_3 tr*

Assignment operator.

Access to predicate objects

The following access functions are provided

Construct_segment_2	*traits.construct_segment_2_object()*
Construct_triangle_2	*traits.construct_triangle_2_object()*
Construct_line_2	*traits.construct_line_2_object()*
Comparison_x_2	*traits.compare_x_2_object()*
Comparison_y_2	*traits.compare_y_2_object()*
Orientation_2	*traits.orientation_2_object()*
Side_of_oriented_circle_2	*traits.side_of_oriented_circle_2_object()*
Compare_distance_2	*traits.compare_distance_2_object()*
Intersect_2	*traits.intersect_2_object()*
Compute_squared_distance_2	*traits.compute_squared_distance_2_object()*

See Also

TriangulationTraits_2
DelaunayTriangulationTraits_2
CGAL::Triangulation_2<Traits,Tds>
CGAL::Delaunay_triangulation_2<Traits,Tds>

CGAL provides also predefined geometric traits class *Triangulation_euclidean_traits_yz_3<K>* and *Triangulation_euclidean_traits_xz_3<K>* to deal with projections on the *xz*- or the *yz*-plane, respectively.

#include <CGAL/Triangulation_euclidean_traits_xz_3.h>
#include <CGAL/Triangulation_euclidean_traits_yz_3.h>

deprecated

CGAL::Triangulation_face_base_2<Traits,Fb>

Class

Definition

The class *Triangulation_face_base_2<Traits,Fb>* is a model for the concept *TriangulationFaceBase_2*. It is the default face base class for basic and Delaunay triangulation.

These default base class can be used directly or can serve as a base to derive other base classes with some additional attribute (a color for example) tuned for specific applications.

Parameters

The first template parameter of *Triangulation_face_base_2<Traits,Fb>* is a geometric traits class. The geometric traits is actually not used by the class.

The second template parameter has to be a model of the concept *TriangulationDSFaceBase_2* and will serve as a base class for *Triangulation_face_base_2<Traits,Fb>* . CGAL provides a default instantiation for this parameter which is *Triangulation_ds_face_base_2<>*.

#include <CGAL/Triangulation_face_base_2.h>

Is Model for the Concepts

TriangulationFaceBase_2

See Also

CGAL::Triangulation_ds_face_base_2<Tds>
CGAL::Triangulation_vertex_base_2<Traits,Vb>
CGAL::Triangulation_2<Traits,Tds>

Class

CGAL::Triangulation_face_base_with_info_2<Info,Traits,Fb>

Definition

The class *Triangulation_face_base_with_info_2<Info,Traits,Fb>* is a model of the concept *TriangulationFaceBase_2* to be plugged into the triangulation data structure of a triangulation class. It provides an easy way to add some user defined information in the faces of a triangulation.

#include <CGAL/Triangulation_face_base_with_info_2.h>

Parameters

The first template argument is the information the user would like to add to a face. It has to be *DefaultConstructible* and *Assignable*.

The second template argument is a geometric traits class and is actually not used in *Triangulation_face_base_with_info_2<Info,Traits,Fb>* .

The third parameter is a face base class from which *Triangulation_face_base_with_info_2<Info,Traits,Fb>* derives.

Inherits From

Fb

Is Model for the Concepts

Because *Triangulation_face_base_with_info_2<Info,Traits,Fb>* derives from the class instantiating its third parameter, it will be a model of the same face base concept as its parameter : *TriangulationFaceBase_2*,
ConstrainedTriangulationFaceBase_2 ,
or *RegularTriangulationFaceBase_2*

Types

typedef Info	*Info;*

Access Functions

const Info&	*f.info() const*	Returns a const reference to the object of type *Info* stored in the face.
Info&	*f.info()*	Returns a reference to the object of type *Info* stored in the face.

See Also

CGAL::Triangulation_face_base_2<Traits,Fb>
CGAL::Constrained_triangulation_face_base_2<Traits,Fb>
CGAL::Regular_triangulation_face_base_2<Traits,Fb>

CGAL::Triangulation_hierarchy_2<Tr>

Class

Definition

The class *Triangulation_hierarchy_2<Tr>* implements a triangulation augmented with a data structure which allows fast point location queries.

The data structure is a hierarchy of triangulations. The triangulation at the lowest level is the original triangulation where operations and point location are to be performed. Then at each succeeding level, the data structure stores a triangulation of a small random sample of the vertices of the triangulation at the preceding level.

Point location is done through a top-down nearest neighbor query. The nearest neighbor query is first performed naively in the top level triangulation. Then, at each following level, the nearest neighbor at that level is found through a linear walk performed from the nearest neighbor found at the preceding level.

Because the number of vertices in each triangulation is only a small fraction of the number of vertices of the preceding triangulation the data structure remains small and achieves fast point location queries on real data. As proved in [Dev98], this structure has an optimal behavior when it is built for Delaunay triangulations. However it can be used as well for other triangulations. The class *Triangulation_hierarchy_2<Tr>* is templated by a parameter which is to be instantiated by anyone of the CGAL triangulation classes.

#include <CGAL/Triangulation_hierarchy_2.h>

Inherits From

Tr

Types

The class *Triangulation_hierarchy_2<Tr>* inherits the types from its base triangulation class *Tr*.

The class *Triangulation_hierarchy_2<Tr>* offers exactly the same functionalities as the triangulation Tr does. Location queries are overloaded to benefit from the data structure. Modifiers (insertion, removal, and displacement) are overloaded to take care of updating the data structure.

Be careful that I/O operations are not overloaded. Writing a *Triangulation_hierarchy_2<Tr>* into a file writes only the lowest level triangulation and drop the hierarchy and reading it from a file results in a triangulation whose efficiency will be that of an ordinary triangulation.

See Also

CGAL::Triangulation_2<Traits,Tds>
CGAL::Delaunay_triangulation_2<Traits,Tds> TriangulationHierarchyVertexBase_2,
CGAL::Triangulation_hierarchy_vertex_base_2<Vb>

Class

CGAL::Triangulation_hierarchy_vertex_base_2<Vb>

Definition

The class *Triangulation_hierarchy_vertex_base_2<Vb>* is designed to be used as a vertex base class of a triangulation plugged into a *Triangulation_hierarchy_2<Tr>*.

It is a model of the concept *TriangulationHierarchyVertexBase_2* which refines the concept *TriangulationVertexBase_2*.

This class is templated by a parameter *Vb* which is to be instantiated by a model of the concept *TriangulationVertexBase_2*. The class *Triangulation_hierarchy_vertex_base_2<Vb>* inherits from the class *Vb*. This design allows to use either the default vertex base class or a user customized vertex base with additional functionalities.

#include <CGAL/Triangulation_hierarchy_2.h>

Is Model for the Concepts

TriangulationHierarchyVertexBase_2

Inherits From

Vb

See Also

TriangulationVertexBase_2
TriangulationHierarchyVertexBase_2
CGAL::Triangulation_vertex_base_2<Traits>

CGAL::Triangulation_vertex_base_2<Traits,Vb>

Definition

The class *Triangulation_vertex_base_2<Traits,Vb>* is the default model for the concept *TriangulationVertexBase_2*.

Triangulation_vertex_base_2<Traits,Vb> can be simply plugged in the triangulation data structure of a triangulation, or used as a base class to derive other base vertex classes tuned for specific applications.

#include <CGAL/Triangulation_vertex_base_2.h>

Parameters

Triangulation_vertex_base_2<Traits,Vb> is templated by a geometric traits class which provide the type *Point*. It is strongly recommended to instantiate this traits class with the model used for the triangulation traits class. This ensures that the point type defined by *Triangulation_vertex_base_2<Traits,Vb>* is the same as the point type defined by the triangulation.

The second template parameter of *Triangulation_vertex_base_2<Traits,Vb>* has to be a model of the concept *TriangulationDSVertexBase_2* By default this parameter is instantiated by *CGAL::Triangulation_ds_vertex_base_2<>*.

Is Model for the Concepts

TriangulationVertexBase_2

Inherits From

Vb

See Also

CGAL::Triangulation_ds_vertex_base_2<Tds>
CGAL::Triangulation_face_base_2<Traits,Fb>
CGAL::Regular_triangulation_vertex_base_2<Traits,Vb>
CGAL::Triangulation_vertex_base_with_info_2<Info,Traits,Vb>

CGAL::Triangulation_vertex_base_with_info_2<Info,Traits,Vb>

Definition

The class *Triangulation_vertex_base_with_info_2<Info,Traits,Vb>* is designed to be used as a base vertex class of a triangulation. It provides an easy way to add some user defined information in vertices.

#include <CGAL/Triangulation_vertex_base_with_info_2.h>

Parameters

The first template parameter is the information the user would like to add to a vertex. It has to be *DefaultConstructible* and *Assignable*.

The second template parameter is the geometric traits class which provides the *Point_2*. It is strongly recommended to instantiate this parameter with the traits class used for the triangulation. This ensures that the point type defined by *Triangulation_vertex_base_with_info_2<Info,Traits,Vb>* matches the point type defined by the triangulation.

The third template parameter is a vertex base class from which *Triangulation_vertex_base_with_info_3* derives. By default this parameter is instantiated by *CGAL::Triangulation_vertex_base_2<Traits>*.

Is Model for the Concepts

TriangulationVertexBaseWithInfo_2
The parameter *Vb* is a model of some vertex base concept. *Triangulation_vertex_base_with_info_2<Info,Traits,Vb>* derives from *Vb* and will be a model of the same vertex base concept : *TriangulationVertexBase_2*, or
RegularTriangulationVertexBase_2.

Types

typedef Info	*Info;*

Access Functions

const Info&	*v.info() const*	Returns a const reference to the object of type *Info* stored in the vertex.
Info&	*v.info()*	Returns a reference to the object of type *Info* stored in the vertex.

See Also

CGAL::Triangulation_face_base_with_info_2<Info,Traits,Fb>
CGAL::Triangulation_vertex_base_2<Traits,Vb>
CGAL::Regular_triangulation_vertex_base_2<Traits,Vb>

CGAL::Weighted_point<Pt,Wt>

Class

Definition

The class *Weighted_point<Pt,Wt>* provides a type associating a point type *Pt* with a weight type *Wt*. It is used in the traits classes *Regular_triangulation_euclidean_traits_2* and *Regular_triangulation_euclidean_traits_3*.

#include <CGAL/Weighted_point.h>

Inherits From

Pt

Types

Pt	*Point;*	The point type
Wt	*Weight;*	The weight type.

Creation

Weighted_point<Pt,Wt> wp(Weighted_point wq);

copy constructor.

Weighted_point<Pt,Wt> wp(Point p=Point(), Weight w= Weight(0));

Weighted_point<Pt,Wt> wp(FT x, FT y);

Constructs the point from x and y coordinates, with a weight of 0. Requires that the ambient dimension be 2.

Weighted_point<Pt,Wt> wp(FT x, FT y, FT z);

Constructs the point from x, y and z coordinates, with a weight of 0. Requires that the ambient dimension be 3.

Access Functions

Point	*wp.point() const*
Weight	*wp.weight() const*

See Also

CGAL::Regular_triangulation_euclidean_traits_2<Rep,Weight>
CGAL::Regular_triangulation_euclidean_traits_3<R,Weight>.

Chapter 38

2D Triangulation Data Structure

Sylvain Pion and Mariette Yvinec

Contents

38.1 Definition

A triangulation data structure is a data structure designed to handle the representation of a two dimensional triangulation. The concept of triangulation data structure was primarily designed to serve as a data structure for CGAL 2D triangulation classes which are triangulations embedded in a plane. However it appears that the concept is more general and can be used for any orientable triangulated surface without boundary, whatever may be the dimensionality of the space the triangulation is embedded in.

38.1.1 A Data Structure Based on Faces and Vertices

The representation of CGAL 2D triangulations is based on faces and vertices, Edges are only implicitly represented trough the adjacency relations between two faces.

The triangulation data structure can be seen as a container for faces and vertices maintaining incidence and adjacency relations among them.

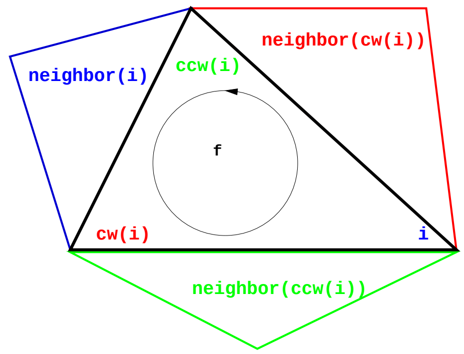


Figure 38.1: Vertices and neighbors.

Each triangular face gives access to its three incident vertices and to its three adjacent faces. Each vertex gives access to one of its incident faces and through that face to the circular list of its incident faces.

The three vertices of a face are indexed with 0, 1 and 2. The neighbors of a face are also indexed with 0,1,2 in such a way that the neighbor indexed by i is opposite to the vertex with the same index. See Figure 38.1, the functions *ccw(i)* and *cw(i)* shown on this figure compute respectively $i+1$ and $i-1$ modulo 3

Each edge has two implicit representations: the edge of a face f which is opposed to the vertex indexed i, can be represented as well as an edge of the *neighbor(i)* of f.

This kind or representation of simplicial complexes extends in any dimension. More precisely, in dimension d, the data structure will explicitly represents cells (i. e. faces of maximal dimension) and vertices (i. e. faces of dimension 0). All faces of dimension between 1 and $d-1$ will have an implicit representation. The 2D triangulation data structure can represent simplicial complexes of dimension 2, 1 or 0.

38.1.2 The Set of Faces and Vertices

The set of faces maintained by a 2D triangulation data structure is such that each edge is incident to two faces. In other words, the set of maintained faces is topologically equivalent to a two-dimensional triangulated sphere.

This rules extends to lower dimensional triangulation data structure arising in degenerate cases or when the triangulations have less than three vertices. A one dimensional triangulation structure maintains a set of vertices and edges which forms a ring topologically equivalent to a 1-sphere.

A zero dimensional triangulation data structure only includes two adjacent vertices that is topologically equivalent to a 0-sphere.

38.2 The Concept of Triangulation Data Structure

A model of *TriangulationDataStructure_2* can be seen has a container for the faces and vertices of the triangulation. This class is also responsible for the combinatorial integrity of the triangulation. This means that the triangulation data structure maintains proper incidence and adjacency relations among the vertices and faces of a triangulation while combinatorial modifications of the triangulation are performed. The term combinatorial modification refers to operations which do not involve any knowledge about the geometric embedding of the triangulation. For example, the insertion of a new vertex in a given face, or in a given edge, the suppression

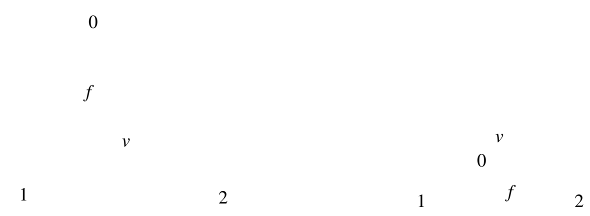


Figure 38.2: Insertion of a new vertex, splitting a face

of a vertex of degree three, the flip of two edge are examples of combinatorial operation performed at the data structure level.

The triangulation data structure is required to provide:

- the types *Vertex* and *Face* for the vertices and faces of the triangulations
- the type *Vertex_handle* and *Face_handle* which are models of the concept *Handle* and through which the vertices and faces are accessed.
- iterators to visit all the vertices, edges and faces of the triangulation,
- circulators to visit all the vertices, edges and faces incident to a given vertex

The triangulation data structure is responsible for the creation and removal of faces and vertices (memory management). It provides function that gives the number of faces, edges and vertices of the triangulation.

The triangulation data structure provides member functions to perform the following combinatorial transformation of the triangulation:

- flip of two adjacent faces,
- addition of a new vertex splitting a given face see Figure 38.2,
- addition of a new vertex splitting a given edge,
- addition of a new vertex raising by one the dimension of a degenerate – lower dimensional triangulation,
- removal of a vertex incident to three faces,
- removal of a vertex lowering the dimension of the triangulation

38.3 The Default Triangulation Data Structure

CGAL provides the class *CGAL::Triangulation_data_structure_2<Vb,Fb>* as a default triangulation data structure.

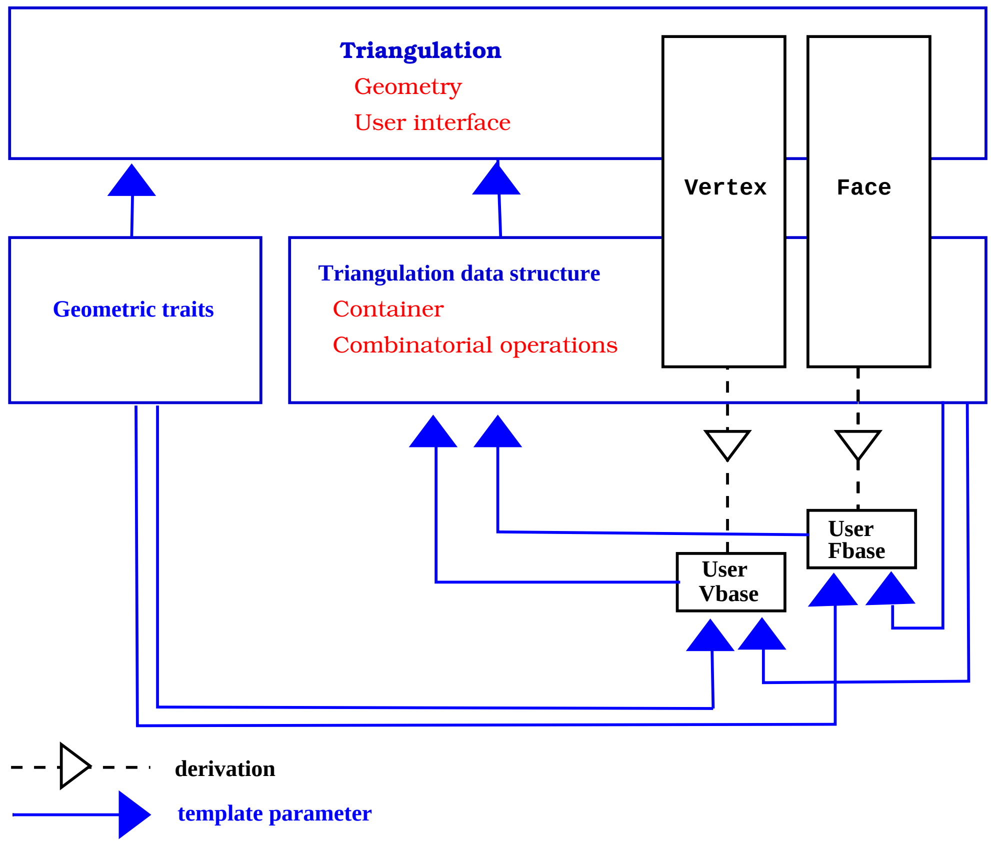


Figure 38.3: The cyclic dependency in triangulations software design.

38.3.1 Flexibility

In order to provide flexibility, the default triangulation data structure is templated by two parameters which stands respectively for a vertex base class and a face base class. The concept *TriangulationDSVertexBase_2* and *TriangulationDSFaceBase_2* describe the requirements for the vertex and face classes of a triangulation data structure.

This design allows the user to plug in the triangulation data structure his own vertex or face classes tuned for his application.

38.3.2 The Cyclic Dependency of Template Parameters

Since adjacency and incidence relation are stored in vertices and faces, the vertex and face classes have to know the types of handles on faces and vertices provided by the triangulation data structure. Therefore, vertex and face classes need to be templated by the triangulation data structure. Because the triangulation data structure is itself templated by the vertex and face classes this induces a cyclic dependency. See figure 38.3.

38.3.3 The Rebind Mechanism

The solution proposed by CGAL to resolve this cyclic dependency is based on a rebind mechanism similar to the mechanism used in the standard allocator class std::allocator. The vertex and face classes plugged in the instantiation of a triangulation data structure are themselves instantiated with a fake data structure. The triangulation data structure will then rebind these classes, plugging itself at the place of the fake data structure, before using them to derive the vertex and face classes. The rebinding is performed through a nested template class *Rebind_TDS* in the vertex and face class, which provide the rebound class as a type called *Other*.

Here is how it works schematically. First, here is the rebinding taking place in the triangulation data structure.

```
template < class Vb, class Fb >
class Triangulation_data_structure
{
  typedef Triangulation_data_structure<Vb,Fb>    Self;

  // Rebind the vertex and face base to the actual TDS (Self).
  typedef typename Vb::template Rebind_TDS<Self>::Other  VertexBase;
  typedef typename Fb::template Rebind_TDS<Self>::Other  FaceBase;

  // ... further internal machinery leads to the final public types:
public:
  typedef ...  Vertex;
  typedef ...  Face;
  typedef ...  Vertex_handle;
  typedef ...  Face_handle;
};
```

Then, here is the vertex class with its nested *Rebind_TDS* template class and its template parameter set by default to an an internal type faking a triangulation data structure.

```
template < class TDS = an internal type faking a triangulation data
structure >
class Vertex_base
{
public:
  template < class TDS2 >
  struct Rebind_TDS {
    typedef Vertex_base<TDS2>    Other;
  };
...
};
```

Imagine an analog *Face_base* class. The triangulation data structure is then instantiate as follows :

```
typedef Triangulation_data_structure< Vertex_base<>, Face_base<> > TDS;
```

38.3.4 Making Use of the Flexibility

There is several possibilities to make use of the flexibility offered by the triangulation data structure.

- First, when the user needs to have, in vertices and faces, additional information which do not depend on types defined by the triangulated data structure, predefined classes *Triangulation_vertex_base_with_info* and *Triangulation_face_base_with_info* can be plugged in. Those classes have a template parameter *Info* to be instantiated by a user defined type. They store a data member of this type and gives access to it.

- Second, the user can derive his own base classes from the default base classes: *Triangulation_ds_vertex_base_2*, and *Triangulation_ds_face_base_2* are the default base classes to be plugged in a triangulation data structure used alone. Triangulation classes requires a data structure in which other base classes have been plugged it. The default base classes for most of the triangulation classes are *Triangulation_vertex_base_2*, and *Triangulation_face_base_2* are the default base classes to be used when the triangulation data structure is plugged in a triangulation class.

  When derivation is used, the rebind mechanism is slightly more involved, because it is necessary to rebind the base class itself. However the user will be able to use in his classes references to types provided by the triangulation data structure. For example,

```
template < class Gt, class Vb = CGAL::Triangulation_vertex_base_2<Gt>
     >
class My_vertex_base
  : public  Vb
{
public :
  template < typename TDS2 >
  struct Rebind_TDS {
    typedef typename Vb::template Rebind_TDS<TDS2>::Other    Vb2;
    typedef My_vertex_base<Gt,Vb2>                           Other;
  };

  typedef typename Vb::Triangulation_data_structure    Tds;
  typedef typename Tds::Vertex_handle                  Vertex_handle;
  ......
};
```

- At last the user can write his own base classes. If the triangulation data structure is used alone, the requirements for the base classes are described by the concepts *TriangulationDSVertexBase_2* and *TriangulationDSFaceBase_2*, documented page 2600 and page 2594. If the triangulation data structure is plugged into a triangulation class, the concepts for the vertex and base classes depends on the triangulation class. The most basic concepts, valid for basic and Delaunay triangulations are *TriangulationVertexBase_2* and *TriangulationFaceBase_2*, documented page 2544 and page 2539.

See section 37.11 for examples of using the triangulation data structure flexibility.

2D Triangulation Data Structure Reference Manual

Sylvain Pion and Mariette Yvinec

The triangulation data structure can be seen as a container for the faces and vertices of a triangulation. This class also takes care of all the combinatorial operations performed on the triangulation.

The class *CGAL::Triangulation_data_structure_2<Vb,Fb>* is a model of the concept *TriangulationDataStructure_2*. which includes the sub-concepts *TriangulationDataStructure_2::Face* and *TriangulationDataStructure_2::Vertex*.

To ensure all the **flexibility** of the triangulation classes, described and in Section 37.11 of Chapter 37, the model *CGAL::Triangulation_data_structure_2<Vb,Fb>* has two templates parameters. The class *CGAL::Triangulation_data_structure_2<Vb,Fb>* derives its *Vertex* and *Face* types from the two template parameters *Vb* and *Fb* respectively.
If the triangulation data structure is used alone, these parameters have to be instantiated by models of the concepts *TriangulationDSFaceBase_2* and *TriangulationDSVertexBase_2*. These concepts are described in this chapter together with their default models *CGAL::Triangulation_ds_face_base_2<Tds>* and *CGAL::Triangulation_ds_vertex_base_2<Tds>*.
If the triangulation data structure is plugged into a triangulation class, the parameters have to be instantiated by models of different refining concepts according to the actual type of the triangulation. These refining concepts and their models are described in Chapter 37.12

38.4 Classified Reference Pages

Concepts

Classes

38.5 Alphabetical List of Reference Pages

TriangulationDataStructure_2

Definition

The concept TriangulationDataStructure_2 describes the requirements for the second template parameter of the basic triangulation class *Triangulation_2<Traits,Tds>* and of all other 2D triangulation classes.

The concept can be seen as a container for the faces and vertices of the triangulation. The concept TriangulationDataStructure_2 includes two sub-concepts *TriangulationDataStructure_2::Vertex* and *TriangulationDataStructure_2::Face*, described respectively page 2602 and page 2597.

The TriangulationDataStructure_2 maintains incidence and adjacency relations among vertices and faces.

Each triangular face gives access to its three incident vertices and to its three adjacent faces. Each vertex gives access to one of its incident faces and through that face to the circular list of its incident faces.

The three vertices of a face are indexed with 0, 1 and 2. The neighbors of a face are also indexed with 0,1,2 in such a way that the neighbor indexed by i is opposite to the vertex with the same index.

Each edge has two implicit representations : the edge of a face f which is opposed to the vertex indexed i, can be represented as well as an edge of the *neighbor(i)* of f. See Figure 37.2

The triangulation data structure is responsible for the combinatorial integrity of the triangulation. This means that the triangulation data structure allows to perform some combinatorial operations on the triangulation and guarantees the maintenance on proper incidence and adjacency relations among the vertices and faces. The term combinatorial operations means that those operations are purely topological and do not depend on the geometric embedding. Insertion of a new vertex in a given face, or in a given edge, suppression of a vertex of degree three, flip of two edges are examples of combinatorial operations.

Types

TriangulationDataStructure_2:: size_type	Size type (unsigned integral type)
TriangulationDataStructure_2:: difference_type	Difference type (signed integral type)

TriangulationDataStructure_2:: Vertex	The vertex type. Requirements for this type are described in concept *TriangulationDataStructure_2::Vertex* page 2602.
TriangulationDataStructure_2:: Face	The face type. Requirements for this type are described in concept *TriangulationDataStructure_2::Face* page 2597.

Vertices and faces are accessed via *Vertex_handle* and *Face_handle*. These types are models of the concept *Handles* which basically supports the two dereference operators * and ->.

TriangulationDataStructure_2:: Vertex_handle	Handle to a vertex
TriangulationDataStructure_2:: Face_handle	Handle to a face.

advanced

TriangulationDataStructure_2:: template <typename Vb2> struct Rebind_vertex

This nested template class allows to get the type of a triangulation data structure that only changes the vertex type. It has to define a type *Other* which is a *rebound* triangulation data structure, that is, the one whose *TriangulationDSVertexBase_2* will be *Vb2*.

TriangulationDataStructure_2:: template <typename Fb2> struct Rebind_face

This nested template class allows to get the type of a triangulation data structure that only changes the face type. It has to define a type *Other* which is a *rebound* triangulation data structure, that is, the one whose *TriangulationDSFaceBase_2* will be *Fb2*.

advanced

typedef std::pair<Face_handle,int> Edge; The edge type. The *Edge(f,i)* is edge common to faces f and *f.neighbor(i)*. It is also the edge joining the vertices *vertex(cw(i))* and *vertex(ccw(i))* of f.

The following iterators allow one to visit all the vertices, edges and faces of a triangulation data structure. They are all bidirectional, non-mutable iterators.

TriangulationDataStructure_2:: Face_iterator
TriangulationDataStructure_2:: Edge_iterator
TriangulationDataStructure_2:: Vertex_iterator

The following circulators allow to visit all the edges or faces incident to a given vertex and all the vertices adjacent to a given vertex. They are all bidirectional and non mutable.

TriangulationDataStructure_2:: Face_circulator
TriangulationDataStructure_2:: Edge_circulator
TriangulationDataStructure_2:: Vertex_circulator

Iterators and circulators are convertible to the corresponding handles, thus they can be passed directly as argument to the functions expecting a handle.

Creation

TriangulationDataStructure_2 tds; default constructor.

TriangulationDataStructure_2 tds(tds1); Copy constructor. All the vertices and faces are duplicated.

TriangulationDataStructure_2& $tds = tds1$

Assignment. All the vertices and faces of *tds1* are duplicated in *tds* . Former faces and vertices of *tds* , if any, are deleted

Vertex_handle *tds.copy_tds(tds1, Vertex_handle v = Vertex_handle())*

tds1 is copied into *tds*. If $v \,!= NULL$, the vertex of *tds* corresponding to *v* is returned, otherwise *Vertex_handle()* is returned.
Precondition: The optional argument *v* is a vertex of *tds1*.

void *tds.swap(& tds1)* Swaps *tds* and *tds1*. Should be preferred to *tds*=*tds1* or *tds*(*tds1*) when *tds1* is deleted after that.

void *tds.clear()* Deletes all faces and all finite vertices.

Access Functions

int *tds.dimension() const* returns the dimension of the triangulation.

size_type
tds.number_of_vertices() const returns the number of vertices in the data structure.

size_type
tds.number_of_faces() const returns the number of two dimensional faces in the data structure.

size_type
tds.number_of_edges() const returns the number of edges in the triangulation data structure.

size_type
tds.number_of_full_dim_faces() const returns the number of full dimensional faces, i.e. faces of dimension equal to the dimension of the triangulation. This is the actual number of faces stored in the triangulation data structure.

advanced

Setting

void *tds.set_dimension(int n)* sets the dimension.

advanced

Queries

bool *tds.is_vertex(Vertex_handle v) const*

returns true if *v* is a vertex of *tds*.

bool *tds.is_edge(Face_handle fh, int i) const*

tests whether *(fh,i)* is an edge of *tds*. Answers *false* when *dimension()* < 1 .

bool *tds.is_edge(Vertex_handle va, Vertex_handle vb) const*

returns true if *va vb* is an edge of *tds*.

bool *tds.is_edge(Vertex_handle va, Vertex_handle vb, Face_handle &fr, int &i) const*

as previous. In addition, if true is returned *fr* and *i* are set such that the pair *(fr,i)* is the description of the ordered edge *va vb*.

bool *tds.is_face(Face_handle fh) const*

tests whether *fh* is a face of *tds*. Answers *false* when *dimension()* < 2 .

bool *tds.is_face(Vertex_handle v1, Vertex_handle v2, Vertex_handle v3) const*

true if there is a face having *v1*, *v2* and*v3* as vertices.

bool *tds.is_face(Vertex_handle v1, Vertex_handle v2, Vertex_handle v3, Face_handle &fr) const*

as above. In addition, if *true* is returned, fr is a pointer to the face with *v1*, *v2* and *v3* as vertices.

Traversing the triangulation

Face_iterator *tds.faces_begin() const* visits all faces

Face_iterator *tds.faces_end() const*

Vertex_iterator *tds.vertices_begin() const*

visits all vertices

Vertex_iterator *tds.vertices_end() const*

Edge_iterator *tds.edges_begin() const*

visits all edges

Edge_iterator *tds.edges_end() const*

Three circulator classes allow to traverse the edges or faces incident to a vertex or the vertices adjacent to this vertex.. A face circulator is invalidated by any modification of the face it points to. An edge circulator is invalidated by any modification of anyone of the two faces incident to the edge pointed to. A vertex circulator that turns around vertex *v* and that has as value a pointer to vertex *w*, is invalidated by any modification of anyone of the two faces incident to *v* and *w*.

Vertex_circulator *tds.incident_vertices(Vertex_handle v, Face_handle f=NULL) const*

Precondition: If the face *f* is given, it has to be incident to be a face of *tds* incident to *v* and the circulator begins with the vertex *f->vertex(ccw(i))* if *i* is the index of *v* in *f*.

Edge_circulator *tds.incident_edges(Vertex_handle v, Face_handle f=NULL) const*

Precondition: If the face *f* is given, it has to be a face of *tds* incident to *v* and the circulator begins with the edge *(f,cw(i))* of *f* if *i* is the index of *v* in *f*.

Face_circulator *tds.incident_faces(Vertex_handle v, Face_handle f=NULL) const*

Precondition: If the face *f* is given, it has to be a face of *tds* incident to *v* and the circulator begins with the face *f*.

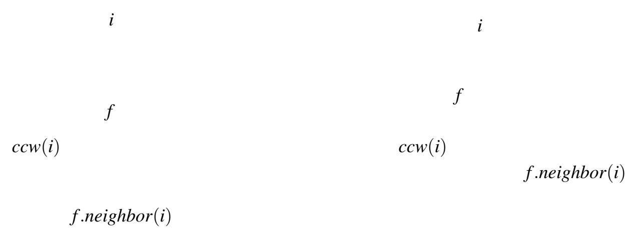


Figure 38.4: Flip.

Vertical_handle *tds.mirror_vertex(Face_handle f, int i) const*

returns vertex of *f->neighbor(i)*.

int *tds.mirror_index(Face_handle f, int i) const*

returns the index of *f* as a neighbor of *f->neighbor(i)*.

Edge *tds.mirror_edge(Edge e) const*

returns the same edge seen from the other adjacent face.

Modifiers

The following modifier member functions guarantee the combinatorial validity of the resulting triangulation.

void *tds.flip(Face_handle f, int i)* exchanges the edge incident to *f* and *f->neighbor(i)* with the other diagonal of the quadrilateral formed by *f* and *f->neighbor(i)*.

Vertex_handle *tds.insert_first()* creates the first vertex and returns a pointer to it.

Vertex_handle *tds.insert_second()* creates the second vertex and returns a pointer to it.

Vertex_handle *tds.insert_in_edge(Face_handle f, int i)* adds a vertex *v* splitting edge *i* of face *f*. Return a pointer to *v*.

Vertex_handle *tds.insert_in_face(Face_handle f)* adds a vertex *v* splitting face *f* in three. Face *f* is modified, two new faces are created. Return a pointer to *v*

Vertex_handle *tds.insert_dim_up(Vertex_handle w, bool orient=true)*

adds a vertex *v*, increasing by one the dimension of the triangulation. Vertex *v* and the existing vertex *w* are linked to all the vertices of the triangulation. The Boolean *orient* decides the final orientation of all faces. A pointer to vertex *v* is returned.

Figure 38.5: Insertion

void *tds.remove_degree_3(Vertex_handle v, Face *f=NULL)*

removes a vertex of degree 3. Two of the incident faces are destroyed, the third one is modified. If parameter f is specified, it has to be a face incident to v and will be the modified face.
Precondition: Vertex v is a finite vertex with degree 3 and, if specified, face f is incident to v.

void *tds.remove_second(Vertex_handle v)* removes the before last vertex.

void *tds.remove_first(Vertex_handle v)* removes the last vertex.

void *tds.remove_dim_down(Vertex_handle v)* removes vertex v incident to all other vertices and decreases by one the dimension of the triangulation.
Precondition: if the dimension is 2, the number of vertices is more than 3, if the dimension is 1, the number of vertices is 2.

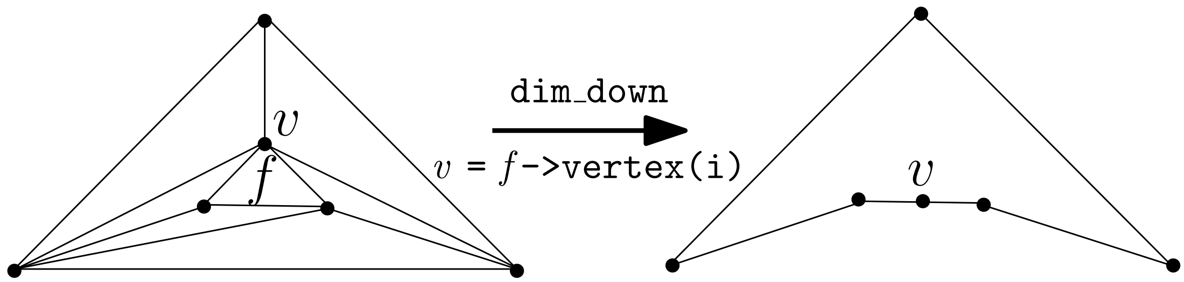


Figure 38.6: From a two-dimensional data structure to a one-dimensional data structure.

The following operation, `dim_down`, is necessary when the displacement of a vertex decreases the dimension of the triangulation.

void *tds.dim_down(Face_handle f, int i)*

The link of a vertex v is formed by the edges disjoint from v that are included in the faces incident to v. When the link of v = *f->vertex(i)* contains all the other vertices of the two-dimensional triangulation data-structure ($\mathbb{S}^2$), *dim_down* crushes the two-dimensional data-structure ($\mathbb{S}^2$) onto the one-dimensional data structure ($\mathbb{S}^1$) formed by the link of v augmented with the vertex v itself; this one is placed on the edge *(f, i)*. (see Fig. 38.6).
Precondition: *dimension()* must be equal to *2*, the degree of *f->vertex(i)* must be equal to the total number of vertices minus 1.

advanced

The following modifiers are required for convenience of the advanced user. They do not guarantee the combinatorial validity of the resulting triangulation.

template< class EdgeIt>
Vertex_handle *tds.star_hole(EdgeIt edge_begin, EdgeIt edge_end)*

creates a new vertex v and use it to star the hole whose boundary is described by the sequence of edges *[edge_begin, edge_end[*. Returns a pointer to the vertex.

template< class EdgeIt, class FaceIt>
Vertex_handle *tds.star_hole(EdgeIt edge_begin, EdgeIt edge_end, FaceIt face_begin, FaceIt face_end)*

same as above, except that, to build the new faces, the algorithm first recycles faces in the sequence *[face_begin, face_end[* and create new ones when the sequence is exhausted.

template< class EdgeIt>
void *tds.star_hole(Vertex_handle v, EdgeIt edge_begin, EdgeIt edge_end)*

uses vertex v to star the hole whose boundary is described by the sequence of edges*[edge_begin, edge_end[*.

template< class EdgeIt, class FaceIt>

void *tds.star_hole(Vertex_handle v,*
EdgeIt edge_begin,
EdgeIt edge_end,
FaceIt face_begin,
FaceIt face_end)

same as above, recycling faces in the sequence *[face_begin, face_end[* .

void *tds.make_hole(Vertex_handle v, List_edges& hole)*

removes the vertex v, and store in *hole* the list of edges on the boundary of the hole.

Vertex_handle *tds.create_vertex()* adds a new vertex.

Face_handle *tds.create_face(Face_handle f1, int i1, Face_handle f2, int i2, Face_handle f3, int i3)*

adds a face which is the neighbor *i1* of *f1*, *i2* of *f2* and *i3* of *f3*.

Face_handle *tds.create_face(Face_handle f1, int i1, Face_handle f2, int i2)*

adds a face which is the neighbor *i1* of *f1*, and the neighbor *i2* of *f2*.

Face_handle *tds.create_face(Face_handle f1, int i1, Vertex_handle v)*

adds a face which is the neighbor *i1* of *f1*, and has *v* as vertex.

Face_handle *tds.create_face(Vertex_handle v1, Vertex_handle v2, Vertex_handle v3)*

adds a face with vertices *v1*, *v2* and *v3*.

Face_handle *tds.create_face(Vertex_handle v1,*
Vertex_handle v2,
Vertex_handle v3,
Face_handle f1,
Face_handle f2,
Face_handle f3)

adds a face with vertices *v1*, *v2* and *v3*, and neighbors *f1*, *f2*, *f3*.

Face_handle *tds.create_face()* adds a face whose vertices and neighbors are set to NULL.

void *tds.delete_face(Face_handle)* deletes a face.

void *tds.delete_vertex(Vertex_handle)*

deletes a vertex.

advanced

Miscellaneous

int *tds.ccw(int i) const* returns $i+1$ modulo 3.
Precondition: $0 \leq i \leq 2$.

int	*tds.cw(int i) const*	returns $i+2$ modulo 3. *Precondition*: $0 \leq i \leq 2$.
bool	*tds.is_valid()*	checks the combinatorial validity of the triangulation: call the *is_valid()* member function for each vertex and each face, checks the number of vertices and the Euler relation between numbers of vertices, faces and edges.
size_type	*tds.degree(Vertex_handle v) const*	Returns the degree of v in the triangulation.

I/O

The information output in the *iostream* is: the dimension, the number of (finite) vertices, the number of (finite) faces. Then comes for each vertex, the non combinatorial information stored in that vertex if any. Then comes for each faces, the indices of its vertices and the non combinatorial information (if any) stored in this face. Then comes for each face again the indices of the neighboring faces. The index of an item (vertex of face) the rank of this item in the output order. When dimension < 2, the same information is output for faces of maximal dimension instead of faces.

void	*tds.file_output(ostream& os, Vertex_handle v = Vertex_handle(), bool skip_first=false)*	writes *tds* into the stream *os*. If v is not a null handle, vertex v is output first or skipped if *skip_first* is true.
Vertex_handle	*tds.file_input(istream& is, bool skip_first=false)*	inputs *tds* from file and returns a pointer to the first input vertex. If *skip_first* is true, it is assumed that the first vertex has been omitted when output.
istream&	*istream& is >> TriangulationDataStructure_3 & tds*	reads a combinatorial triangulation from *is* and assigns it to *tds*
ostream&	*ostream& os << TriangulationDataStructure_3 tds*	writes *tds* into the stream *os*

Has Models

CGAL::Triangulation_data_structure_2<Vb,Fb>

See Also

TriangulationDataStructure_2::Face
TriangulationDataStructure_2::Vertex
CGAL::Triangulation_2<Traits,Tds>

Concept

TriangulationDSFaceBase_2

Definition

The concept TriangulationDSFaceBase_2 describes the requirements for the base face of a *Triangulation_data_structure_2<Vb,Fb>*.

Note that if the *Triangulation_data_structure_2* is plugged into a triangulation class, the face base class may have additional geometric requirements depending on the triangulation class.

At the base level, (see Sections 37.3 and 38.3), a face stores handles on its three vertices and on the three neighboring faces. The vertices and neighbors are indexed 0,1 and 2. Neighbor i lies opposite to vertex i.

Since the *Triangulation_data_structure_2* is the class which defines the handle types, the face base class has to be somehow parameterized by the triangulation data structure. But since the *Triangulation_data_structure_2* itself is parameterized by the face and vertex base classes, there is a cycle in the definition of these classes. In order to break the cycle, the base classes for faces and vertices which are plugged in to instantiate a *Triangulation_data_structure_2* use a *void* as triangulation data structure parameter. Then, the *Triangulation_data_structure_2* uses a *rebind* mechanism (similar to the one specified in *std::allocator*) in order to plug itself as parameter in the face and vertex base classes. This mechanism requires that the base class provides a templated nested class *Rebind_TDS* that itself provides the subtype *Rebind_TDS<TDS2>::Other* which is the *rebound* version of the base class. This *rebound* base class is the class that the *Triangulation_data_structure_2* actually uses as a base class for the class *Triangulation_data_structure_2::Face*.

Types

The concept TriangulationDSFaceBase_2 has to provide the following types.

TriangulationDSFaceBase_2:: template <typename TDS2> struct Rebind_TDS

This nested template class has to define a type *Other* which is the *rebound* face base, where the *Triangulation_data_structure_2* is actually plugged in. This type *Other* will be the actual base of the class *Triangulation_data_structure_2::Face*.

typedef TriangulationDataStructure_2 *Triangulation_data_structure;*
typedef TriangulationDataStructure_2::Vertex_handle *Vertex_handle;*
typedef TriangulationDataStructure_2::Face_handle *Face_handle;*

Creation

TriangulationDSFaceBase_2 f; default constructor.

TriangulationDSFaceBase_2 f(Vertex_handle v0, Vertex_handle v1, Vertex_handle v2);

Initializes the vertices with *v0, v1, v2* and the neighbors with *Face_handle()*.

TriangulationDSFaceBase_2 f(Vertex_handle v0,
Vertex_handle v1,
Vertex_handle v2,
Face_handle n0,
Face_handle n1,
Face_handle n2)

initializes the vertices with *v0,v1, v2* and the neighbors with *n0, n1, n2*.

Access Functions

int	*f.dimension()*	returns the dimension.
Vertex_handle	*f.vertex(int i) const*	returns the vertex *i* of *f*. *Precondition*: $0 \leq i \leq 2$.
bool	*f.has_vertex(Vertex_handle v)*	returns true if *v* is a vertex of *f*.
bool	*f.has_vertex(Vertex_handle v, int& i) const*	as above, and sets *i* to the index of *v* in *f*.
int	*f.index(Vertex_handle v) const*	returns the index of *v* in *f*.
Face_handle	*f.neighbor(int i) const*	returns the neighbor *i* of *f*. *Precondition*: $0 \leq i \leq 2$.
bool	*f.has_neighbor(Face_handle n)*	returns true if *n* is a neighbor of *f*.
bool	*f.has_neigbor(Face_handle n, int& i) const*	as above, and sets i to the index of *n* in *f*.
int	*f.index(const Face_handle n) const*	returns the index of neighbor *n* in *f*.

Setting

void	*f.set_vertex(int i, Vertex_handle v)*	sets vertex *i* to *v*. *Precondition*: $0 \leq i \leq 2$.
void	*f.set_vertices()*	sets the vertices to *Vertex_handle()*.
void	*f.set_vertices(Vertex_handle v0, Vertex_handle v1, Vertex_handle v2)*	sets the vertices.
void	*f.set_neighbor(int i, Face_handle n)*	sets neighbors *i* to *n*. *Precondition*: $0 \leq i \leq 2$.
void	*f.set_neighbors()*	sets the neighbors to *Face_handle()*.
void	*f.set_neighbors(Face_handle n0, Face_handle n1, Face_handle n2)*	sets the neighbors.

Orientation

void	*f.reorient()*	Changes the orientation of *f* by exchanging *vertex(0)* with *vertex(1)* and *neighbor(0)* with *neighbor(1)*.

void	f*.ccw_permute()*	preforms a counterclockwise permutation of the vertices and neighbors of f.
void	f*.cw_permute()*	preforms a clockwise permutation of the vertices and neighbors of f.

Checking

bool	f*.is_valid(bool verbose = false) const*	performs any required test on a face. If *verbose* is set to *true*, messages are printed to give a precise indication of the kind of invalidity encountered.

Various

*void**	f*.for_compact_container() const*
*void**&	f*.for_compact_container()*

These member functions are required by *Triangulation_data_structure_2* because it uses *Compact_container* to store its faces. See the documentation of *Compact_container* for the exact requirements.

Has Models

CGAL::Triangulation_ds_face_base_2<Tds>
CGAL::Triangulation_face_base_2<Traits,Fb>
CGAL::Regular_triangulation_face_base_2<Traits,Fb>
CGAL::Constrained_triangulation_face_base_2<Traits,Fb>
CGAL::Triangulation_face_base_with_info_2<Info,Traits,Fb>

See Also

TriangulationDSVertexBase_2
TriangulationDataStructure_2::Face
TriangulationFaceBase_2
Triangulation_data_structure_2<Vb,Fb>

TriangulationDataStructure_2::Face

Concept

Definition

The concept TriangulationDataStructure_2::Face describes the types used to store the faces face class of a *TriangulationDataStructure_2*, see page 2585. A TriangulationDataStructure_2::Face stores three pointers to its three vertices and three pointers to its three neighbors. The vertices are indexed 0,1, and 2 in counterclockwise order. The neighbor indexed i lies opposite to vertex i.

In degenerate cases, when the triangulation data structure stores a simplicial complex of dimension 0 and 1, the type TriangulationDataStructure_2::Face is used to store the faces of maximal dimension of the complex : i.e. a vertex in dimension 0, an edge in dimension 1. Only vertices and neighbors with index 0 are set in the first case, only vertices and neighbors with index 0 or 1 are set in the second case.

Types

The class TriangulationDataStructure_2::Face defines the same types as the triangulation data structure except the iterators and the circulators.

Creation

The methods *create_face* and *delete_face()* have to be used to define new faces and to delete non longer used faces.

Vertex Access Functions

Vertex_handle	*f.vertex(int i) const*	returns the vertex i of f. *Precondition*: $0 \leq i \leq 2$.
int	*f.index(Vertex_handle v) const*	returns the index of vertex v in f. *Precondition*: v is a vertex of f
bool	*f.has_vertex(Vertex_handle v) const*	returns *true* if v is a vertex of f.
bool	*f.has_vertex(Vertex_handle v, int& i) const*	returns *true* if v is a vertex of f, and computes the index i of v in f.

Neighbor Access Functions

The neighbor with index i is the neighbor which is opposite to the vertex with index i.

Face_handle	*f.neighbor(int i) const*	returns the neighbor i of f. *Precondition*: $0 \leq i \leq 2$.
int	*f.index(Face_handle n) const*	returns the index of face n. *Precondition*: n is a neighbor of f.

bool	*f.has_neighbor(Face_handle n) const*	
		returns *true* if n is a neighbor of f.
bool	*f.has_neighbor(Face_handle n, int& i) const*	
		returns *true* if n is a neighbor of f, and compute the index i of n.

advanced

Setting

void	*f.set_vertex(int i, Vertex_handle v)*	sets vertex i to be v. *Precondition*: $0 \leq i \leq 2$.
void	*f.set_neighbor(int i, Face_handle n)*	sets neighbor i to be n. *Precondition*: $0 \leq i \leq 2$.
void	*f.set_vertices()*	sets the vertices pointers to *NULL*.
void	*f.set_vertices(Vertex_handle v0, Vertex_handle v1, Vertex_handle v2)*	
		sets the vertices pointers.
void	*f.set_neighbors()*	sets the neighbors pointers to *NULL*.
void	*f.set_neighbors(Face_handle n0, Face_handle n1, Face_handle n2)*	
		sets the neighbors pointers.

Checking

bool	*f.is_valid() const*	returns *true* if the function *is_valid()* of the base class returns *true* and if, for each index i, $0 \leq i < 3$, face f is a neighbor of its neighboring face *neighbor(i)* and shares with this neighbor the vertices *cw(i)* and *ccw(i)* in correct reverse order.

advanced

Miscellaneous

int	*f.ccw(int i) const*	Returns $i+1$ modulo 3. *Precondition*: $0 \leq i \leq 2$.
int	*f.cw(int i) const*	Returns $i+2$ modulo 3. *Precondition*: $0 \leq i \leq 2$.

I/O

istream&	*istream& is* >> *& f*	Inputs any non combinatorial information possibly stored in the face.
ostream&	*ostream& os* << *f*	Outputs any non combinatorial information possibly stored in the face.

See Also

TriangulationDataStructure_2,
TriangulationDataStructure_2::Vertex,
TriangulationFaceBase_2.

Concept

TriangulationDSVertexBase_2

Definition

The concept TriangulationDSVertexBase_2 describes the requirements for the vertex base class of a *Triangulation_data_structure_2<Vb,Fb>*.

Note that if the *Triangulation_data_structure_2* is plugged into a triangulation class, the vertex base class has additional geometric requirements depending on the triangulation class.

At the base level, provides access to one of its incident face through a *Face_handle*.

Since the *Triangulation_data_structure_2* is the class which defines the handle types, the vertex base class has to be somehow parameterized by the triangulation data structure. But since the *Triangulation_data_structure_2* itself is parameterized by the face and vertex base classes, there is a cycle in the definition of these classes. In order to break the cycle, the base classes for faces and vertices which are plugged in to instantiate a *Triangulation_data_structure_2* use a *void* as triangulation data structure parameter. Then, the *Triangulation_data_structure_2* uses a *rebind* mechanism (similar to the one specified in *std::allocator*) in order to plug itself as parameter in the face and vertex base classes. This mechanism requires that the base class provides a templated nested class *Rebind_TDS* that itself provides the subtype *Rebind_TDS<TDS2>::Other* which is the *rebound* version of the base class. This *rebound* base class is the class that the *Triangulation_data_structure_2* actually uses as a base class for the class of *Triangulation_data_structure_2::Vertex*.

Refines

TriangulationDataStructure_2::Vertex

Types

The concept TriangulationDSVertexBase_2 has to provide the following types.

TriangulationDSVertexBase_2:: template <typename TDS2> struct Rebind_TDS

This nested template class has to define a type *Other* which is the *rebound* vertex base , where the actual *Triangulation_data_structure_2* is plugged in. This type *Other* will be the actual base of the class *Triangulation_data_structure_2::Vertex*.

typedef TriangulationDataStructure_2	*Triangulation_data_structure;*
typedef TriangulationDataStructure_2::Vertex_handle	*Vertex_handle;*
typedef TriangulationDataStructure_2::Face_handle	*Face_handle;*

Creation

TriangulationDSVertexBase_2 v;	default constructor.
TriangulationDSVertexBase_2 v(Face_handle f);	Constructs a vertex pointing to face *f*.

Various

*void**	*v.for_compact_container() const*
void&*	*v.for_compact_container()*

These member functions are required by *Triangulation_data_structure_2* because it uses *Compact_container* to store its faces. See the documentation of *Compact_container* for the exact requirements.

Has Models

CGAL::Triangulation_ds_vertex_base_2<Tds>
CGAL::Triangulation_vertex_base_2<Traits,Vb>
CGAL::Regular_triangulation_vertex_base_2<Traits,Vb>
CGAL::Triangulation_hierarchy_vertex_base_2<Vb>
CGAL::Triangulation_vertex_base_with_info_2<Info,Traits,vb>

See Also

TriangulationVertexBase_2
TriangulationDSFaceBase_2
TriangulationFaceBase_2
TriangulationDataStructure_2::Vertex
Triangulation_data_structure_2<Vb,Fb>

Concept

TriangulationDataStructure_2::Vertex

Definition

The concept TriangulationDataStructure_2::Vertex describes the type used by a *TriangulationDataStructure_2* to store the vertices, see page 2585.

Some of the requirements listed below are of geometric nature and are *optional* when using the triangulation data structure class alone. They became required when the triangulation data structure is plugged into a triangulation.

Types

The class TriangulationDataStructure_2::Vertex defines the same types as the triangulation data structure except the iterators.

TriangulationDataStructure_2::Vertex:: Point *Optional for the triangulation data structure used alone.*

Creation

In order to obtain new vertices or destruct unused vertices, the user must call the *create_vertex()* and *delete_vertex()* methods of the triangulation data structure.

Access Functions

Point	*v.point() const*	returns the geometric information of *v*.
Face_handle	*v.face() const*	returns a face of the triangulation having *v* as vertex.

advanced

Setting

void	*v.set_point(Point p)*	sets the geometric information to *p*.
void	*v.set_face(Face_handle f)*	sets the incident face to *f*.

advanced

Checking

bool *v.is_valid(bool verbose = false) const*

Checks the validity of the vertex. Must check that its incident face has this vertex. The validity of the base vertex is also checked. When *verbose* is set to *true*, messages are printed to give a precise indication on the kind of invalidity encountered.

I/O

istream&	*istream& is >> & v*	Inputs the non-combinatorial information possibly stored in the vertex.
ostream&	*ostream& os << v*	Outputs the non combinatorial operation possibly stored in the vertex.

Has Models

CGAL::Triangulation_ds_vertex_2<Vb,Fb>

See Also

TriangulationDataStructure_2
TriangulationDataStructure_2::Face

Class

CGAL::Triangulation_data_structure_2<Vb,Fb>

Definition

The class *Triangulation_data_structure_2<Vb,Fb>* is a model for the *TriangulationDataStructure_2* concept. It can be used to represent an orientable 2D triangulation embedded in a space of any dimension.

The vertices and faces are stored in two nested containers, which are implemented using *CGAL::Compact_container*. The class may offer some flexibility for the choice of container in the future, in the form of additional template parameters.

#include <CGAL/Triangulation_data_structure_2.h>

Is Model for the Concepts

TriangulationDataStructure_2

advanced

In addition to the interface documented in the concept, the class offers the following types and functions.

Types

typedef CGAL::Compact_container<Vertex>

Vertex_range;	Vertex container type.

typedef CGAL::Compact_container<Face>

Face_range;	Face container type.

Operations

Face_range&	*tds.faces() const*	Returns a reference to the container of faces.
Face_range&	*tds.faces()*	Returns a reference to the container of faces.
Vertex_range&	*tds.vertices() const*	Returns a reference to the container of vertices.
Vertex_range&	*tds.vertices()*	Returns a reference to the container of vertices.

advanced

Modifiers

In addition to the modifiers required by the *TriangulationDataStructure_2* concept, the *Triangulation_data_structure_2<Vb,Fb>* class supports also the modifiers below. Note also that the modifiers below guarantee the combinatorial validity of the resulting data structure.

Vertex_handle	*tds.join_vertices(Face_handle f, int i)*	Joins the vertices that are endpoints of the edge *(f,i)*. It returns a vertex handle to common vertex (see Fig. 38.7). *Precondition*: *f* must be different from *Face_handle()* and *i* must be *0*, *1* or *2*.

Vertex_handle	*tds.join_vertices(Edge e)*	Joins the vertices that are endpoints of the edge *e*. It returns a vertex handle to common vertex.
Vertex_handle	*tds.join_vertices(Edge_iterator eit)*	Joins the vertices that are endpoints of the edge **eit*. It returns a vertex handle to common vertex.
Vertex_handle	*tds.join_vertices(Edges_circulator ec)*	Joins the vertices that are endpoints of the edge **ec*. It returns a vertex handle to common vertex.

boost::tuples::tuple<Vertex_handle, Vertex_handle, Face_handle, Face_handle>

tds.split_vertex(Vertex_handle v, Face_handle f1, Face_handle f2)

Splits the vertex *v* into two vertices *v1* and *v2*. The common faces *f* and *g* of *v1* and *v2* are created after (in the counter-clockwise sense) the faces *f1* and *f2*. The 4-tuple *(v1,v2,f,g)* is returned (see Fig. 38.7). *Precondition*: *dimension()* must be equal to *2*, *f1* and *f2* must be different from *Face_handle()* and *v* must be a vertex of both *f1* and *f2*.

Vertex_handle	*tds.insert_degree_2(Face_handle f, int i)*	Inserts a degree two vertex and two faces adjacent to it that have two common edges. The edge defined by the face handle *f* and the integer *i* is duplicated. It returns a handle to the vertex created (see Fig. 38.8).
void	*tds.remove_degree_2(Vertex_handle v)*	Removes a degree 2 vertex and the two faces adjacent to it. The two edges of the star of *v* that are not incident to it are collapsed (see Fig. 38.8). *Precondition*: The degree of *v* must be equal to 2.

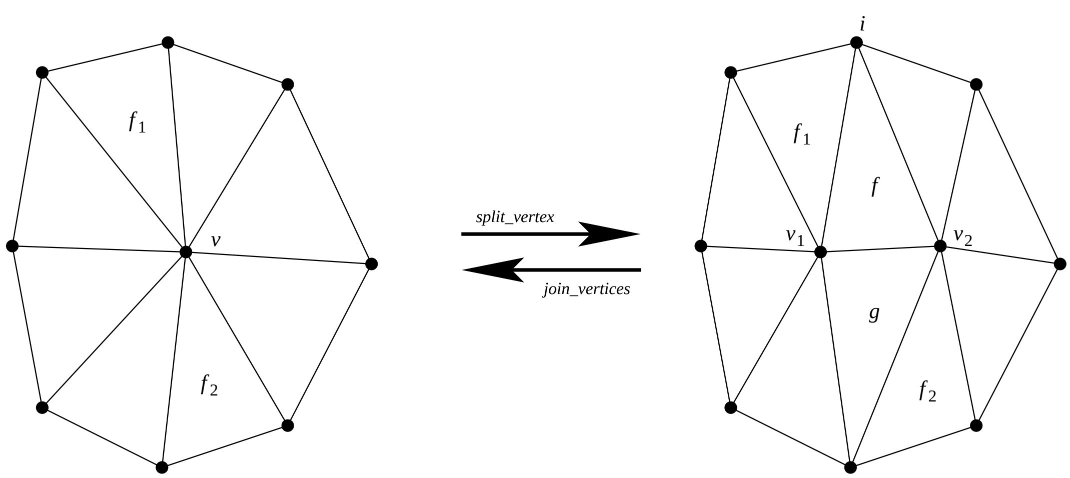


Figure 38.7: The join and split operations.

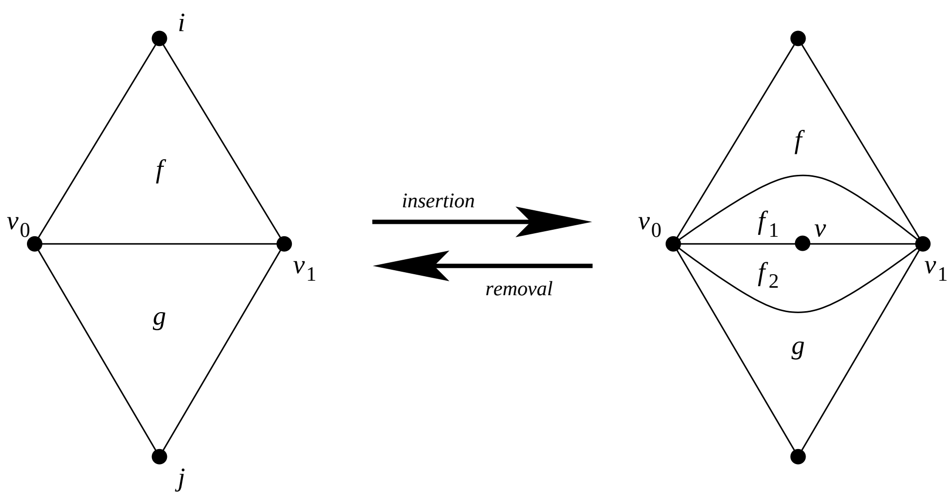


Figure 38.8: Insertion and removal of degree 2 vertices.

CGAL::Triangulation_ds_face_base_2<Tds>

Class

Definition

The class *Triangulation_ds_face_base_2<Tds>* is a model for the concept *TriangulationDSFaceBase_2* to be used by *Triangulation_data_structure_2*.

#include <CGAL/Triangulation_ds_face_base_2.h>

Is Model for the Concepts

TriangulationDSFaceBase_2

See Also

CGAL::Triangulation_face_base_2<Traits,Fb>
CGAL::Triangulation_ds_vertex_base_2<Tds>

Class

CGAL::Triangulation_ds_vertex_base_2<Tds>

Definition

The class *Triangulation_ds_vertex_base_2* can be used as the base vertex for a *Triangulation_data_structure_2*, it is a model of the concept *TriangulationDSVertexBase_2*.

This base class can be used directly or can serve as a base to derive other base classes with some additional attributes (a color for example) tuned for a specific application.

Note that if the *Triangulation_data_structure_2* is used as a parameter of a geometric triangulation, there are additional geometric requirements to be fulfilled by the vertex base class, and *Triangulation_ds_vertex_base_2* cannot be plugged in.

#include <CGAL/Triangulation_ds_vertex_base_2.h>

Is Model for the Concepts

TriangulationDSVertexBase_2

See Also

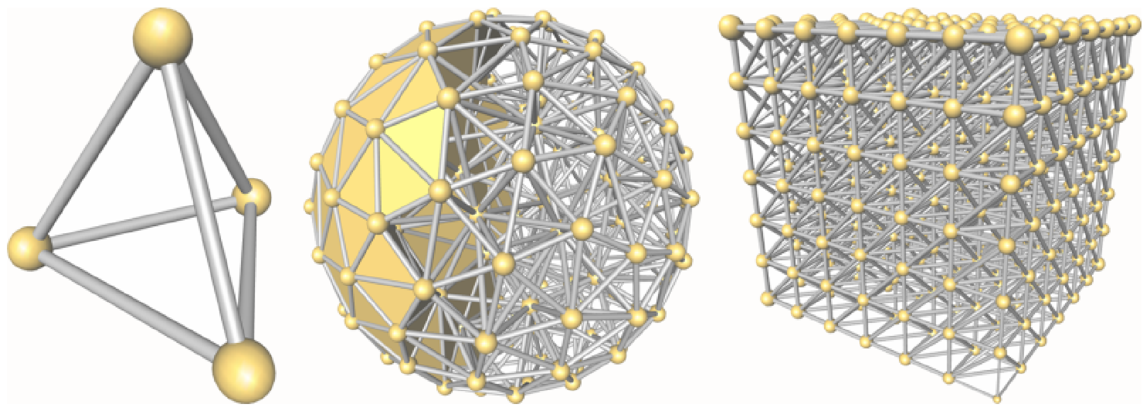
CGAL::Triangulation_vertex_base_2<Traits,Vb>
CGAL::Triangulation_ds_face_base_2<Tds>

Chapter 39

3D Triangulations

Sylvain Pion and Monique Teillaud

Contents



The basic 3D-triangulation class of CGAL is primarily designed to represent the triangulations of a set of points A in $\mathbb{R}^3$. It is a partition of the convex hull of A into tetrahedra whose vertices are the points of A. Together with the unbounded cell having the convex hull boundary as its frontier, the triangulation forms a partition of $\mathbb{R}^3$. Its cells (3-faces) are such that two cells either do not intersect or share a common facet (2-face), edge (1-face) or vertex (0-face).

39.1 Representation

In order to deal only with tetrahedra, which is convenient for many applications, the unbounded cell can be subdivided into tetrahedra by considering that each convex hull facet is incident to an *infinite cell* having as fourth vertex an auxiliary vertex called the *infinite vertex*. In that way, each facet is incident to exactly two cells and special cases at the boundary of the convex hull are simple to deal with.

The class *Triangulation_3<TriangulationTraits_3,TriangulationDataStructure_3>* of CGAL implements this point of view and therefore considers the triangulation of the set of points as a set of finite and infinite tetrahedra. Notice that the infinite vertex has no significant coordinates and that no geometric predicate can be applied on it.

A triangulation is a collection of vertices and cells that are linked together through incidence and adjacency relations. Each cell gives access to its four incident vertices and to its four adjacent cells. Each vertex gives access to one of its incident cells.

The four vertices of a cell are indexed with 0, 1, 2 and 3 in positive orientation, the positive orientation being defined by the orientation of the underlying Euclidean space $\mathbb{R}^3$ (see Figure 39.1). The neighbors of a cell are also indexed with 0, 1, 2, 3 in such a way that the neighbor indexed by i is opposite to the vertex with the same index.

As in the underlying combinatorial triangulation (see Chapter 40), edges (1-faces) and facets (2-faces) are not explicitly represented: a facet is given by a cell and an index (the facet i of a cell c is the facet of c that is opposite to the vertex with index i) and an edge is given by a cell and two indices (the edge *(i,j)* of a cell c is the edge whose endpoints are the vertices of c with indices i and j). See Figure 40.1.

Degenerate Dimensions The class *Triangulation_3* can also deal with triangulations whose dimension d is less than 3. A triangulation of a set of points in $\mathbb{R}^d$ covers the whole space $\mathbb{R}^d$ and consists of cells having $d+1$ vertices: some of them are infinite, they are obtained by linking the additional infinite vertex to each facet of the convex hull of the points.

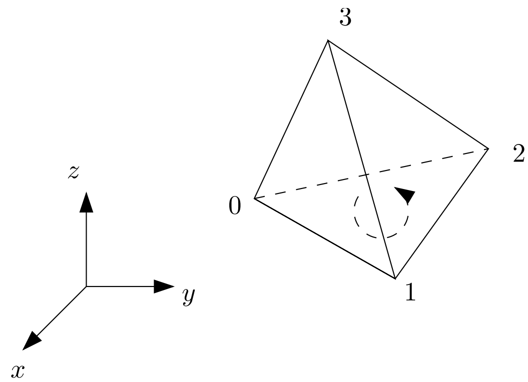


Figure 39.1: Orientation of a cell (3-dimensional case).

- *dimension 2:* when a triangulation only contains coplanar points (which is the case when there are only three points), it consists of triangular faces.
- *dimension 1:* the triangulation contains only collinear points (which is the case when there are only two points), it consists of edges.
- *dimension 0:* the triangulation contains only one finite point.
- *dimension -1:* this is a convention to handle the case when the only vertex of the triangulation is the infinite one.

The same cell class is used in all cases: triangular faces in 2D can be considered as degenerate cells, having only three vertices (resp. neighbors) numbered $(0,1,2)$; edges in 1D have only two vertices (resp. neighbors) numbered 0 and 1.

The implicit representation of facets (resp. edges) still holds for degenerate dimensions (*i.e.* dimensions < 3): in dimension 2, each cell has only one facet of index 3, and 3 edges $(0,1)$, $(1,2)$ and $(2,0)$; in dimension 1, each cell has one edge $(0,1)$.

Validity A triangulation of $\mathbb{R}^3$ is said to be *locally valid* iff

(a)-(b) Its underlying combinatorial graph, the triangulation data structure, is *locally valid* (see Section 40.1 of Chapter 40)
(c) Any cell has its vertices ordered according to positive orientation. See Figure 39.1.

When the triangulation is degenerated into a triangulation of dimension 2, the geometric validity reduces to:

(c-2D) For any two adjacent triangles (u,v,w_1) and (u,v,w_2) with common edge (u,v), w_1 and w_2 lie on opposite sides of (u,v) in the plane.

When all the points are collinear, this condition becomes:

(c-1D) For any two adjacent edges (u,v) and (v,w), u and w lie on opposite sides of the common vertex v on the line.

The *is_valid()* method provided in *Triangulation_3* checks the local validity of a given triangulation. This does not always ensure global validity [MNS+96, DLPT98] but it is sufficient for practical cases.

39.2 Delaunay Triangulation

The class *Delaunay_triangulation_3* represents a three-dimensional Delaunay triangulation.

Delaunay triangulations have the specific *empty sphere property*, that is, the circumscribing sphere of each cell of such a triangulation does not contain any other vertex of the triangulation in its interior. These triangulations are uniquely defined except in degenerate cases where five points are co-spherical. Note however that the CGAL implementation computes a unique triangulation even in these cases.

This implementation is fully dynamic: it supports insertions of points, vertex removals and displacements of points.

39.3 Regular Triangulation

The class *Regular_triangulation_3* implements incremental regular triangulations, also known as weighted Delaunay triangulations.

Let $S^{(w)}$ be a set of weighted points in $\mathbb{R}^3$. Let $p^{(w)} = (p, w_p), p \in \mathbb{R}^3, w_p \in \mathbb{R}$ and $z^{(w)} = (z, w_z), z \in \mathbb{R}^3, w_z \in \mathbb{R}$ be two weighted points. A weighted point $p^{(w)} = (p, w_p)$ can also be seen as a sphere of center p and radius $\sqrt{w_p}$. The *power product* between $p^{(w)}$ and $z^{(w)}$ is defined as

$$\Pi(p^{(w)}, z^{(w)}) = \|p - z\|^2 - w_p - w_z$$

where $\|p - z\|$ is the Euclidean distance between p and z. $p^{(w)}$ and $z^{(w)}$ are said to be *orthogonal* iff $\Pi(p^{(w)}, z^{(w)}) = 0$ (see Figure 39.2).

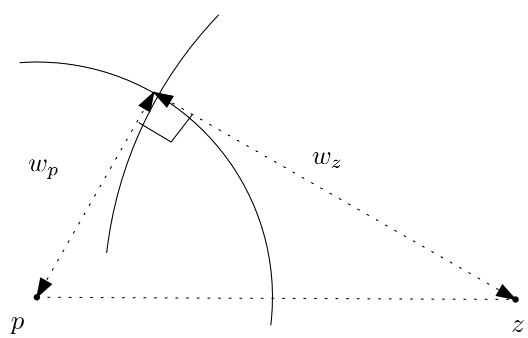


Figure 39.2: Orthogonal weighted points (picture in 2D).

Four weighted points have a unique common orthogonal weighted point called the *power sphere*. The weighted point orthogonal to three weighted points in the plane defined by these three points is called the *power circle*. The *power segment* will denote the weighted point orthogonal to two weighted points on the line defined by these two points.

A sphere $z^{(w)}$ is said to be *regular* if $\forall p^{(w)} \in S^{(w)}, \Pi(p^{(w)}, z^{(w)}) \geq 0$.

A triangulation of $S^{(w)}$ is *regular* if the power spheres of all simplices are regular.

The regular triangulation of $S^{(w)}$ is in fact the projection onto $\mathbb{R}^3$ of the convex hull of the four-dimensional points $(p, \|p - O\|^2 - w_p)$, for $p^{(w)} = (p, w_p) \in S^{(w)}$. Note that all points of $S^{(w)}$ do not necessarily appear as vertices of the regular triangulation. To know more about regular triangulations, see for example [ES96].

When all weights are 0, power spheres are nothing more than circumscribing spheres, and the regular triangulation is exactly the Delaunay triangulation.

The implementation of 3D regular triangulation supports insertions of weighted points, and vertex removals. Displacements are not supported in the current implementation.

39.4 Software Design

The main classes *Triangulation_3*, *Delaunay_triangulation_3* and *Regular_triangulation_3* are connected to each other by the derivation diagram shown in Figure 39.3. This diagram also shows another class: *Triangulation_utils_3* (page 2743), which provides a set of tools operating on the indices of vertices in cells.

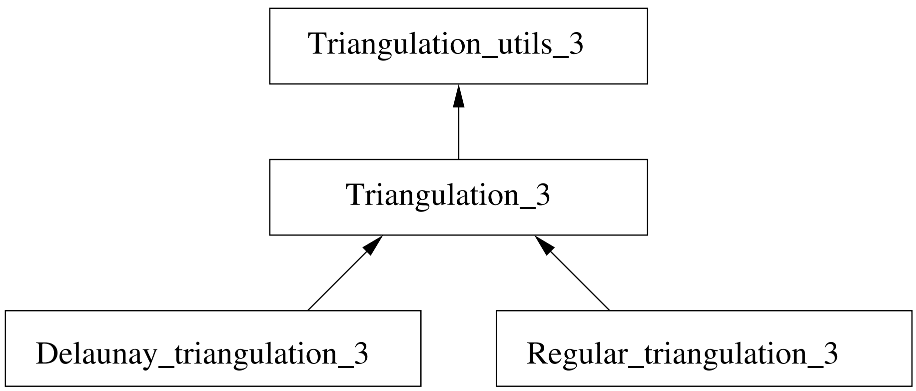


Figure 39.3: Derivation diagram of the 3D triangulation classes.

The three main classes (*Triangulation_3*, *Delaunay_triangulation_3* and *Regular_triangulation_3*) provide high-level geometric functionality such as location of a point in the triangulation [DPT02], insertion and possibly removal of a point [DT03], and are responsible for the geometric validity. They are built as layers on top of a triangulation data structure, which stores their combinatorial structure. This separation between the geometry and the combinatorics is reflected in the software design by the fact that these three triangulation classes take the following template parameters :

- the **geometric traits** class, which provides the type of points to use as well as the elementary operations on them (predicates and constructions). The concepts for these parameters are described in more details in Section 39.4.1 and in page 2672.
- the **triangulation data structure** class, which stores their combinatorial structure, described in Section 40.2 of Chapter 40.
- the **location policy** tag, which is supported only by the Delaunay triangulation class, described in Section 39.4.3.

39.4.1 The Geometric Traits Parameter

The first template parameter of the triangulation class *Triangulation_3<TriangulationTraits_3, TriangulationDataStructure_3>* is the geometric traits class, described by the concept *TriangulationTraits_3*. It must define the types of the geometric objects (points, segments, triangles and tetrahedra) forming the triangulation together with a few geometric predicates on these objects: orientation in space, orientation in case of coplanar points, order of collinear points.

In addition to the requirements described before, the geometric traits class of *Delaunay_triangulation_3* must define predicates to test for the *empty sphere property*. It is described by the concept *DelaunayTriangulationTraits_3*, which refines *TriangulationTraits_3*.

The kernels provided by CGAL: *Cartesian*, *Homogeneous*, *Simple_cartesian*, *Simple_homogeneous* and *Filtered_kernel* can all be used as models for the geometric traits parameter. They supply the user with all the functionalities described for the concepts *TriangulationTraits_3* (page 2672) and *DelaunayTriangulationTraits_3* (page 2674). In addition, the predefined kernels *Exact_predicates_inexact_constructions_kernel* (page **??**) and *Exact_predicates_exact_constructions_kernel* (page **??**) can also be used, the latter being recommended when the dual construction is used.

In order to be used as the traits class for *Regular_triangulation_3*, a class must provide functions to compute the *power tests* (see Section 39.3). *Regular_triangulation_euclidean_traits_3<K,Weight>* is a traits class designed to be used by the class *Regular_triangulation_3<RegularTriangulationTraits_3, TriangulationDataStructure_3>*. It provides *Weighted_point*, a class for weighted points needed by the regular triangulation, which derives from the three dimensional point class *K::Point_3*. It supplies the user with all the functionalities described for the concept *RegularTriangulationTraits_3* (page 2677). It can be used as a traits class for *Regular_triangulation_3<RegularTriangulationTraits_3, TriangulationDataStructure_3>*.

Note that for regular triangulations, plugging a filtered kernel such as *Exact_predicates_inexact_constructions_kernel* or *Exact_predicates_exact_constructions_kernel* in *Regular_triangulation_euclidean_traits_3<K,Weight>* will provide exact and efficient filtered predicates.

39.4.2 The Triangulation Data Structure Parameter

The second template parameter of the main classes (*Triangulation_3*, *Delaunay_triangulation_3* and *Regular_triangulation_3*) is a triangulation data structure class. This class can be seen as a container for the cells and vertices maintaining incidence and adjacency relations (see Chapter 40). A model of this triangulation data structure is *Triangulation_data_structure_3* (page 2739), and it is described by the *TriangulationDataStructure_3* concept (page 2715). This model is itself parameterized by a vertex base and a cell base classes, which gives the possibility to customize the vertices and cells used by the triangulation data structure, and hence by the geometric triangulation using it. Depending on the kind of triangulation used, the requirements on the vertex and cell base classes vary, and are expressed by various concepts, following the refinement diagram shown in Figure 39.4.

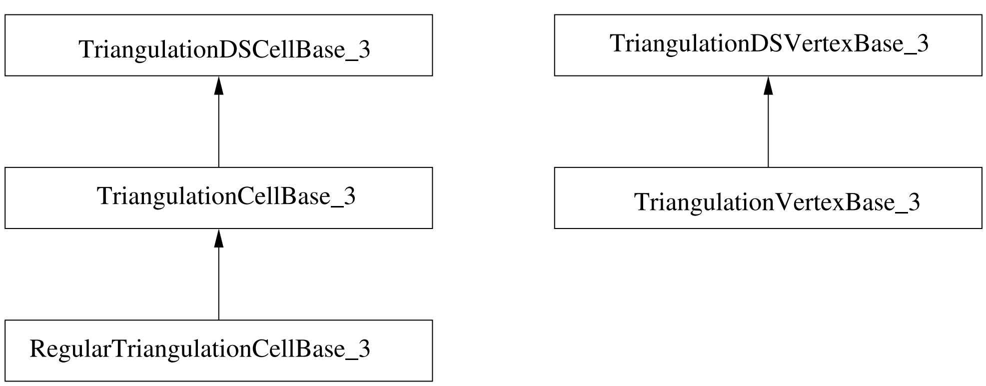


Figure 39.4: Concepts refinement hierarchy for the vertex and cell base classes parameters.

A default value for the triangulation data structure parameter is provided in all the triangulation classes, so it need not be specified by the user unless he wants to use a different triangulation data structure or a different vertex or cell base class.

39.4.3 The Location Policy Parameter

The Delaunay triangulation class supports an optional feature which maintains an additional data structure for fast point location queries. The fast location policy should be used when the user inserts points in a random order or needs to do many unrelated queries. If the user is able to give a good hint to help the point location of its queries (and its newly inserted points), then it should prefer the default policy. In such a case where good hints are provided, the default policy save some memory (few percents), and is faster. Notice that if points are not inserted one by one, but as a range, then a good hint is automatically computed using spatial sort.

Reading Section 39.6 on complexity and performance can help making an informed choice for this parameter.

The point location strategy can be selected with the third template argument of *Delaunay_triangulation_3*, *LocationPolicy*, which enables a fast point location data structure when set to *Fast_location*. By default, it uses *Compact_location*.

Note that you can specify the *LocationPolicy* parameter without specifying the triangulation data structure, in case you are fine with the default there. In this case, the *LocationPolicy* appears as a second parameter after the geometric traits.[1]

The *Fast_location* policy is implemented using a hierarchy of triangulations; it changes the behavior of functions *locate*, *insert*, *move*, and *remove*. As proved in [Dev02], this structure has an optimal behavior when it is built for Delaunay triangulations.

In this setting, if you build a triangulation by iteratively inserting points, you should try to shuffle the points beforehand, as the time complexity is guaranteed only for a randomized order. For example, inserting points in lexicographic order is typically much slower. Note that this shuffling is performed internally by the constructor taking a range of points.

Prior to CGAL 3.6, this functionality was available through the *Triangulation_hierarchy_3* class, which is now deprecated.

39.4.4 Flexibility of the Design

In order to satisfy as many uses as possible, a design has been selected that allows to exchange different parts to meet the users' needs, while still re-using a maximum of the provided functionalities. We have already seen that the main triangulation classes are parameterized by a geometric traits class and a triangulation data structure (TDS), so that each of them can be interchanged with alternate implementations.

The most useful flexibility is the ability given to the user to add his own data in the vertices and cells by providing his own vertex and cell base classes to *Triangulation_data_structure_3*. The Figure 39.5 shows in more detail the flexibility that is provided, and the place where the user can insert his own vertex and/or cell base classes.

The design of the triangulation data structure gives the possibility to store any kind of data, including handles (an entity akin to pointers) directly in the vertex and cell base classes.

To do so, there are three possibilities. The simplest one is to use the class *Triangulation_vertex_base_with_info_3*, and this approach is illustrated in a following subsection 39.5.2. The most complicated one, and probably

[1] The mechanism used behind the scenes to allow this syntactical convenience is called *deduced parameters*.

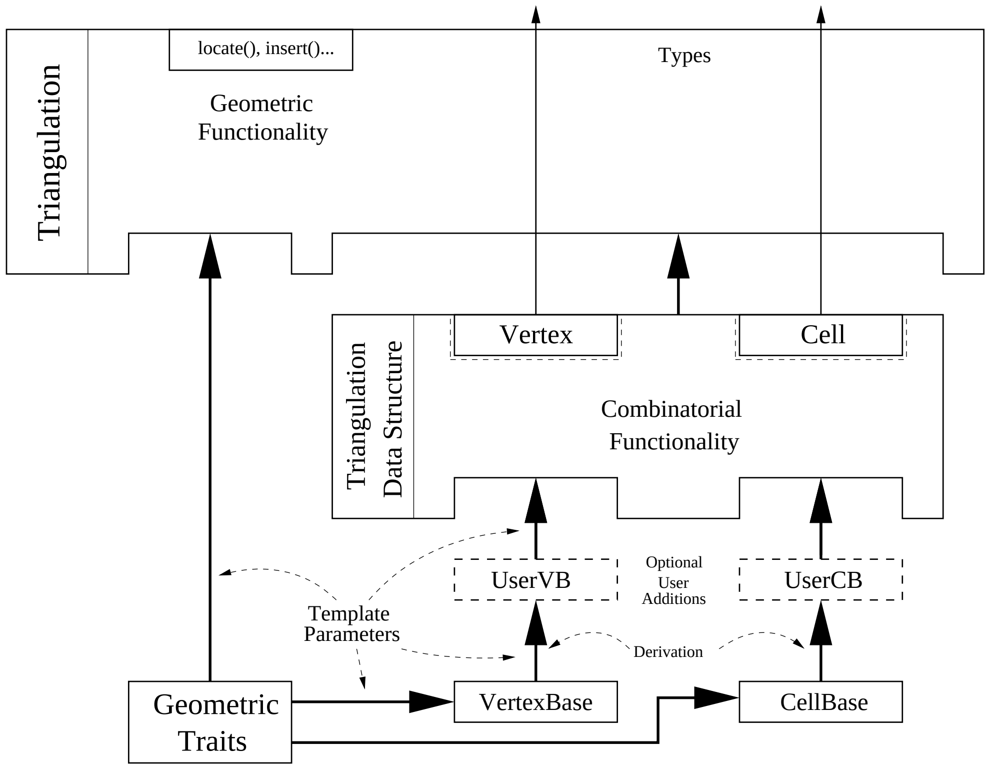


Figure 39.5: Triangulation software design.

useless for almost all cases, is to write a vertex base class from scratch, following the documented requirements. This is mostly useless because most of the time it is enough to derive from the models that CGAL provides, and add the desired features. In this case, when the user needs to access some type that depends on the triangulation data structure (typically handles), then he should write something like:

```
...
template < class GT, class Vb = Triangulation_vertex_base<GT> >
class My_vertex
  : public Vb
{
public:
  typedef typename Vb::Point           Point;
  typedef typename Vb::Cell_handle     Cell_handle;

  template < class TDS2 >
  struct Rebind_TDS {
    typedef typename Vb::template Rebind_TDS<TDS2>::Other  Vb2;
    typedef My_vertex<GT, Vb2>                             Other;
  };

  My_vertex() {}
  My_vertex(const Point&p)                : Vb(p) {}
  My_vertex(const Point&p, Cell_handle c) : Vb(p, c) {}
...
};
... // The rest has not changed
```

The situation is exactly similar for cell base classes. Section 40.2 provides more detailed information.

39.5 Examples

39.5.1 Basic Example

This example shows the incremental construction of a 3D triangulation, the location of a point and how to perform elementary operations on indices in a cell. It uses the default parameter of the *Triangulation_3* class.

```
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/Triangulation_3.h>

#include <iostream>
#include <fstream>
#include <cassert>
#include <list>
#include <vector>

typedef CGAL::Exact_predicates_inexact_constructions_kernel K;

typedef CGAL::Triangulation_3<K>      Triangulation;

typedef Triangulation::Cell_handle    Cell_handle;
typedef Triangulation::Vertex_handle  Vertex_handle;
typedef Triangulation::Locate_type    Locate_type;
typedef Triangulation::Point          Point;

int main()
{
  // construction from a list of points :
  std::list<Point> L;
  L.push_front(Point(0,0,0));
  L.push_front(Point(1,0,0));
  L.push_front(Point(0,1,0));

  Triangulation T(L.begin(), L.end());

  Triangulation::size_type n = T.number_of_vertices();

  // insertion from a vector :
  std::vector<Point> V(3);
  V[0] = Point(0,0,1);
  V[1] = Point(1,1,1);
  V[2] = Point(2,2,2);

  n = n + T.insert(V.begin(), V.end());

  assert( n == 6 );       // 6 points have been inserted
  assert( T.is_valid() ); // checking validity of T

  Locate_type lt;
  int li, lj;
```

```
  Point p(0,0,0);
  Cell_handle c = T.locate(p, lt, li, lj);
  // p is the vertex of c of index li :
  assert( lt == Triangulation::VERTEX );
  assert( c->vertex(li)->point() == p );

  Vertex_handle v = c->vertex( (li+1)&3 );
  // v is another vertex of c
  Cell_handle nc = c->neighbor(li);
  // nc = neighbor of c opposite to the vertex associated with p
  // nc must have vertex v :
  int nli;
  assert( nc->has_vertex( v, nli ) );
  // nli is the index of v in nc

  std::ofstream oFileT("output",std::ios::out);
  // writing file output;
  oFileT << T;

  Triangulation T1;
  std::ifstream iFileT("output",std::ios::in);
  // reading file output;
  iFileT >> T1;
  assert( T1.is_valid() );
  assert( T1.number_of_vertices() == T.number_of_vertices() );
  assert( T1.number_of_cells() == T.number_of_cells() );

  return 0;
}
```

File: examples/Triangulation_3/simple_triangulation_3.cpp

39.5.2 Changing the Vertex Base

The following two examples show how the user can plug his own vertex base in a triangulation. Changing the cell base is similar.

Adding a Color

When the user doesn't need to add a type in a vertex which depends on the *TriangulationDataStructure_3* (e.g. a *Vertex_handle* or *Cell_handle*), then he can use the *Triangulation_vertex_base_with_info_3* class to add his own information easily in the vertices. The example below shows how to add a *CGAL::Color* this way.

```
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/Delaunay_triangulation_3.h>
#include <CGAL/Triangulation_vertex_base_with_info_3.h>
#include <CGAL/IO/Color.h>

typedef CGAL::Exact_predicates_inexact_constructions_kernel K;
```

```
typedef CGAL::Triangulation_vertex_base_with_info_3<CGAL::Color, K> Vb;
typedef CGAL::Triangulation_data_structure_3<Vb>                    Tds;
typedef CGAL::Delaunay_triangulation_3<K, Tds>
   Delaunay;

typedef Delaunay::Point   Point;

int main()
{
  Delaunay T;

  T.insert(Point(0,0,0));
  T.insert(Point(1,0,0));
  T.insert(Point(0,1,0));
  T.insert(Point(0,0,1));
  T.insert(Point(2,2,2));
  T.insert(Point(-1,0,1));

  // Set the color of finite vertices of degree 6 to red.
  Delaunay::Finite_vertices_iterator vit;
  for (vit = T.finite_vertices_begin(); vit != T.finite_vertices_end();
   ++vit)
      if (T.degree(vit) == 6)
          vit->info() = CGAL::RED;

  return 0;
}
```

File: examples/Triangulation_3/color.cpp

Adding Handles

When the user needs to add a type in a vertex which depends on the *TriangulationDataStructure_3* (e.g. a *Vertex_handle* or *Cell_handle*), then he has to derive his own vertex base class, as the following example shows.

```
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/Delaunay_triangulation_3.h>
#include <CGAL/Triangulation_vertex_base_3.h>

template < class GT, class Vb = CGAL::Triangulation_vertex_base_3<GT> >
class My_vertex_base
  : public Vb
{
public:
  typedef typename Vb::Vertex_handle  Vertex_handle;
  typedef typename Vb::Cell_handle    Cell_handle;
  typedef typename Vb::Point          Point;

  template < class TDS2 >
  struct Rebind_TDS {
    typedef typename Vb::template Rebind_TDS<TDS2>::Other  Vb2;
    typedef My_vertex_base<GT, Vb2>                        Other;
```

```
  };

  My_vertex_base() {}

  My_vertex_base(const Point& p)
    : Vb(p) {}

  My_vertex_base(const Point& p, Cell_handle c)
    : Vb(p, c) {}

  Vertex_handle   vh;
  Cell_handle     ch;
};


typedef CGAL::Exact_predicates_inexact_constructions_kernel K;

typedef CGAL::Triangulation_data_structure_3<My_vertex_base<K> >    Tds;
typedef CGAL::Delaunay_triangulation_3<K, Tds>
   Delaunay;

typedef Delaunay::Vertex_handle    Vertex_handle;
typedef Delaunay::Point            Point;

int main()
{
  Delaunay T;

  Vertex_handle v0 = T.insert(Point(0,0,0));
  Vertex_handle v1 = T.insert(Point(1,0,0));
  Vertex_handle v2 = T.insert(Point(0,1,0));
  Vertex_handle v3 = T.insert(Point(0,0,1));
  Vertex_handle v4 = T.insert(Point(2,2,2));
  Vertex_handle v5 = T.insert(Point(-1,0,1));

  // Now we can link the vertices as we like.
  v0->vh = v1;
  v1->vh = v2;
  v2->vh = v3;
  v3->vh = v4;
  v4->vh = v5;
  v5->vh = v0;

  return 0;
}
```

File: examples/Triangulation_3/adding_handles_3.cpp

39.5.3 Setting Information While Inserting a Range of Points

The most efficient method to insert (weighted) points in a Delaunay (or regular) triangulation is to provide an iterator range over (weighted) points to the insert function. However, an iterator range of (weighted) points does not allow to set different information to each vertex. To solve this problem, in the case the vertex type of the triangulation is a model of the concept *TriangulationVertexBaseWithInfo_3* (such as *Triangulation_vertex_base_with_info_3*), we provide three examples doing the same operation: set an unsigned integer as the information of each vertex. The value of this unsigned integer is the initial order of the corresponding point given in the range.

Using an Iterator Over Pairs

Each point and its information are gathered into a pair. We provide the constructor of the triangulation (which is calling the *insert* function) with a range of such pairs.

```
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/Delaunay_triangulation_3.h>
#include <CGAL/Triangulation_vertex_base_with_info_3.h>
#include <vector>

typedef CGAL::Exact_predicates_inexact_constructions_kernel         K;
typedef CGAL::Triangulation_vertex_base_with_info_3<unsigned, K>    Vb;
typedef CGAL::Triangulation_data_structure_3<Vb>                    Tds;
typedef CGAL::Delaunay_triangulation_3<K, Tds>
    Delaunay;
typedef Delaunay::Point
    Point;

int main()
{
  std::vector< std::pair<Point,unsigned> > points;
  points.push_back( std::make_pair(Point(0,0,0),0) );
  points.push_back( std::make_pair(Point(1,0,0),1) );
  points.push_back( std::make_pair(Point(0,1,0),2) );
  points.push_back( std::make_pair(Point(0,0,1),3) );
  points.push_back( std::make_pair(Point(2,2,2),4) );
  points.push_back( std::make_pair(Point(-1,0,1),5) );


  Delaunay T( points.begin(),points.end() );

  CGAL_assertion( T.number_of_vertices() == 6 );

  // check that the info was correctly set.
  Delaunay::Finite_vertices_iterator vit;
  for (vit = T.finite_vertices_begin(); vit != T.finite_vertices_end();
   ++vit)
    if( points[ vit->info() ].first != vit->point() ){
      std::cerr << "Error different info" << std::endl;
      exit(EXIT_FAILURE);
    }
  std::cout << "OK" << std::endl;

  return 0;
}
```

File: examples/Triangulation_3/info_insert_with_pair_iterator.cpp

Using the Boost Zip Iterator

Information and points are in separate containers. We use *boost::zip_iterator* to provide an iterator gathering them.

```
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/Delaunay_triangulation_3.h>
#include <CGAL/Triangulation_vertex_base_with_info_3.h>
#include <boost/iterator/zip_iterator.hpp>
#include <vector>

typedef CGAL::Exact_predicates_inexact_constructions_kernel         K;
typedef CGAL::Triangulation_vertex_base_with_info_3<unsigned, K>    Vb;
typedef CGAL::Triangulation_data_structure_3<Vb>                    Tds;
typedef CGAL::Delaunay_triangulation_3<K, Tds>
   Delaunay;
typedef Delaunay::Point
   Point;

int main()
{

  std::vector<unsigned> indices;
  indices.push_back(0);
  indices.push_back(1);
  indices.push_back(2);
  indices.push_back(3);
  indices.push_back(4);
  indices.push_back(5);

  std::vector<Point> points;
  points.push_back(Point(0,0,0));
  points.push_back(Point(1,0,0));
  points.push_back(Point(0,1,0));
  points.push_back(Point(0,0,1));
  points.push_back(Point(2,2,2));
  points.push_back(Point(-1,0,1));



  Delaunay T( boost::make_zip_iterator(boost::make_tuple(
   points.begin(),indices.begin() )),
              boost::make_zip_iterator(boost::make_tuple(
   points.end(),indices.end() ) )  );

  CGAL_assertion( T.number_of_vertices() == 6 );


  // check that the info was correctly set.
  Delaunay::Finite_vertices_iterator vit;
  for (vit = T.finite_vertices_begin(); vit != T.finite_vertices_end();
   ++vit)
    if( points[ vit->info() ] != vit->point() ){
      std::cerr << "Error different info" << std::endl;
```

```
      exit(EXIT_FAILURE);
    }

  return 0;
}
```

File: examples/Triangulation_3/info_insert_with_zip_iterator.cpp

Using the Boost Transform Iterator

We define a functor *Auto_count* used together with *boost::transform_iterator* to set the order of each point in the range. Note that this is correct because the iterator is dereferenced only once per point during the insertion.

```
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/Delaunay_triangulation_3.h>
#include <CGAL/Triangulation_vertex_base_with_info_3.h>
#include <boost/iterator/transform_iterator.hpp>
#include <vector>

typedef CGAL::Exact_predicates_inexact_constructions_kernel         K;
typedef CGAL::Triangulation_vertex_base_with_info_3<unsigned, K>    Vb;
typedef CGAL::Triangulation_data_structure_3<Vb>                    Tds;
typedef CGAL::Delaunay_triangulation_3<K, Tds>
   Delaunay;
typedef Delaunay::Point
   Point;

//a functor that returns a std::pair<Point,unsigned>.
//the unsigned integer is incremented at each call to
//operator()
struct Auto_count : public std::unary_function<const
   Point&,std::pair<Point,unsigned> >{
  mutable unsigned i;
  Auto_count() : i(0){}
  std::pair<Point,unsigned> operator()(const Point& p) const {
    return std::make_pair(p,i++);
  }
};

int main()
{
  std::vector<Point> points;
  points.push_back(Point(0,0,0));
  points.push_back(Point(1,0,0));
  points.push_back(Point(0,1,0));
  points.push_back(Point(0,0,1));
  points.push_back(Point(2,2,2));
  points.push_back(Point(-1,0,1));


  Delaunay T(
   boost::make_transform_iterator(points.begin(),Auto_count()),
```

```
                boost::make_transform_iterator(points.end(),  Auto_count()
   )  );

  CGAL_assertion( T.number_of_vertices() == 6 );

  // check that the info was correctly set.
  Delaunay::Finite_vertices_iterator vit;
  for (vit = T.finite_vertices_begin(); vit != T.finite_vertices_end();
   ++vit)
    if( points[ vit->info() ] != vit->point() ){
      std::cerr << "Error different info" << std::endl;
      exit(EXIT_FAILURE);
    }
  std::cout << "OK" << std::endl;

  return 0;
}
```

File: examples/Triangulation_3/info_insert_with_transform_iterator.cpp

39.5.4 The Simplex Class

The triangulation defines a *Simplex* class that represents a simplex (vertex, edge, facet or cell). This example demonstrates how simplices can be stored in a set.

```
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/Triangulation_3.h>

#include <iostream>
#include <fstream>
#include <list>
#include <set>

typedef CGAL::Exact_predicates_inexact_constructions_kernel K;

typedef CGAL::Triangulation_3<K>      Triangulation;

typedef Triangulation::Finite_vertices_iterator Finite_vertices_iterator;
typedef Triangulation::Finite_edges_iterator Finite_edges_iterator;
typedef Triangulation::Finite_facets_iterator Finite_facets_iterator;
typedef Triangulation::Finite_cells_iterator Finite_cells_iterator;
typedef Triangulation::Simplex        Simplex;
typedef Triangulation::Locate_type    Locate_type;
typedef Triangulation::Point          Point;

int main()
{
  // construction from a list of points :
  std::list<Point> L;
  L.push_front(Point(0,0,0));
  L.push_front(Point(1,0,0));
  L.push_front(Point(0,1,0));
```

```
  L.push_front(Point(0,1,1));

  Triangulation T(L.begin(), L.end());

  std::set<Simplex> simplices;

  Finite_vertices_iterator vit = T.finite_vertices_begin();
  simplices.insert(Simplex(vit));

  Finite_cells_iterator cit = T.finite_cells_begin();
  simplices.insert(Simplex(cit));

  Finite_edges_iterator eit = T.finite_edges_begin();
  simplices.insert(Simplex(*eit));

  Finite_facets_iterator fit = T.finite_facets_begin();
  simplices.insert(Simplex(*fit));


  for (std::set<Simplex>::iterator it = simplices.begin();
       it != simplices.end(); it++) {
    std::cout << it->dimension() << std::endl;
  }

  return 0;
}
```

File: examples/Triangulation_3/simplex.cpp

39.5.5 Fast Point Location for Delaunay Triangulations

```
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/Delaunay_triangulation_3.h>
#include <CGAL/Random.h>

#include <vector>
#include <cassert>

typedef CGAL::Exact_predicates_inexact_constructions_kernel K;
typedef CGAL::Delaunay_triangulation_3<K, CGAL::Fast_location> Delaunay;
typedef Delaunay::Point Point;

int main()
{
  // generating points on a grid.
  std::vector<Point> P;

  for (int z=0 ; z<20 ; z++)
    for (int y=0 ; y<20 ; y++)
      for (int x=0 ; x<20 ; x++)
          P.push_back(Point(x,y,z));
```

```
  // building their Delaunay triangulation.
  Delaunay T(P.begin(), P.end());

  assert( T.number_of_vertices() == 8000 );

  // performing nearest vertex queries to a series of random points,
  // which is a case where the Fast_location policy is beneficial.
  for (int i=0; i<10000; ++i)
    T.nearest_vertex(Point(CGAL::default_random.get_double(0, 20),
                           CGAL::default_random.get_double(0, 20),
                           CGAL::default_random.get_double(0, 20)));

  return 0;
}
```

File: examples/Triangulation_3/fast_location_3.cpp

39.5.6 Finding the Cells in Conflict with a Point in a Delaunay Triangulation

```
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/Delaunay_triangulation_3.h>
#include <CGAL/point_generators_3.h>

#include <vector>
#include <cassert>

typedef CGAL::Exact_predicates_inexact_constructions_kernel K;

typedef CGAL::Delaunay_triangulation_3<K>         Delaunay;
typedef Delaunay::Point                           Point;
typedef Delaunay::Cell_handle                     Cell_handle;
typedef Delaunay::Facet                           Facet;

int main()
{
  Delaunay T;
  CGAL::Random_points_in_sphere_3<Point> rnd;

  // First, make sure the triangulation is 3D.
  T.insert(Point(0,0,0));
  T.insert(Point(1,0,0));
  T.insert(Point(0,1,0));
  T.insert(Point(0,0,1));

  assert(T.dimension() == 3);

  // Inserts 100 random points if and only if their insertion
  // in the Delaunay tetrahedralization conflicts with
  // an even number of cells.
  for (int i = 0; i != 100; ++i) {
    Point p = *rnd++;
```

```
    // Locate the point
    Delaunay::Locate_type lt;
    int li, lj;
    Cell_handle c = T.locate(p, lt, li, lj);
    if (lt == Delaunay::VERTEX)
      continue; // Point already exists

    // Get the cells that conflict with p in a vector V,
    // and a facet on the boundary of this hole in f.
    std::vector<Cell_handle> V;
    Facet f;

    T.find_conflicts(p, c,
                     CGAL::Oneset_iterator<Facet>(f), // Get one boundary
     facet
                     std::back_inserter(V));          // Conflict cells
   in V

    if ((V.size() & 1) == 0)  // Even number of conflict cells ?
      T.insert_in_hole(p, V.begin(), V.end(), f.first, f.second);
  }

  std::cout << "Final triangulation has " << T.number_of_vertices()
            << " vertices." << std::endl;

  return 0;
}
```

File: examples/Triangulation_3/find_conflicts_3.cpp

39.5.7 Regular Triangulation

This example shows the building of a regular triangulation. In this triangulation, points have an associated weight, and some points can be hidden and do not result in vertices in the triangulation. Another difference is that a specific traits class has to be used (at least at the moment).

```
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/Regular_triangulation_3.h>
#include <CGAL/Regular_triangulation_euclidean_traits_3.h>
#include <cassert>
#include <vector>

typedef CGAL::Exact_predicates_inexact_constructions_kernel K;

typedef CGAL::Regular_triangulation_euclidean_traits_3<K>  Traits;

typedef Traits::RT                                          Weight;
typedef Traits::Bare_point                                  Point;
typedef Traits::Weighted_point
   Weighted_point;

typedef CGAL::Regular_triangulation_3<Traits>               Rt;
```

```
typedef Rt::Vertex_iterator
    Vertex_iterator;
typedef Rt::Vertex_handle
    Vertex_handle;

int main()
{
  // generate points on a 3D grid
  std::vector<Weighted_point> P;

  int number_of_points = 0;

  for (int z=0 ; z<5 ; z++)
    for (int y=0 ; y<5 ; y++)
      for (int x=0 ; x<5 ; x++) {
          Point p(x, y, z);
          Weight w = (x+y-z*y*x)*2.0; // let's say this is the weight.
          P.push_back(Weighted_point(p, w));
          ++number_of_points;
      }

  Rt T;

  // insert all points in a row (this is faster than one insert() at a
    time).
  T.insert (P.begin(), P.end());

  assert( T.is_valid() );
  assert( T.dimension() == 3 );

  std::cout << "Number of vertices : " << T.number_of_vertices() <<
    std::endl;

  // removal of all vertices
  int count = 0;
  while (T.number_of_vertices() > 0) {
      T.remove (T.finite_vertices_begin());
      ++count;
  }

  assert( count == number_of_points );

  return 0;
}
```

File: examples/Triangulation_3/regular_3.cpp

39.6 Complexity and Performance

In 3D, the worst case complexity of a triangulation is quadratic in the number of points. For Delaunay triangulations, this bound is reached in cases such as points equally distributed on two non-coplanar lines. However, the good news is that, in many cases, the complexity of a Delaunay triangulation is linear or close to linear in the number of points. Several articles [Dwy89, Eri02, AAD07, AB03, ABL03] have proved such good complexity bounds for specific point distributions, such as points distributed on surfaces under some conditions.

39.6.1 Running Time

There are several algorithms provided in this package. We will focus here on the following ones and give practical numbers on their efficiency :

- construction of a triangulation from a range of points,
- location of a point (using the *locate* function),
- removal of a vertex (using the *remove* function).

We will use the following types of triangulations, using *Exact_predicates_inexact_constructions_kernel* as geometric traits (combined with *Regular_triangulation_euclidean_traits_3* in the weighted case) :

- **Delaunay** : *Delaunay_triangulation_3*
- **Delaunay - Fast location** : *Delaunay_triangulation_3* with *Fast_location*
- **Regular** : *Regular_triangulation_3* (default setting : memorize hidden points)
- **Regular - No hidden points** : *Regular_triangulation_3* with hidden points discarded (using *Triangulation_cell_base_3* instead of *Regular_triangulation_cell_base_3*).

Figure 39.6 shows, for all these types of triangulations, the times in seconds taken to build a triangulation from a given number of points, then the average time to perform one point location in triangulations of various sizes, and the average time to perform one vertex removal (which is largely independent on the size of the triangulation).

The data sets used here are points randomly distributed in the unit cube (the coordinates are generated using the `drand48` C function). In the weighted case, the weights are all zero, which means that there are actually no hidden points during execution.

The measurements have been performed using CGAL 3.6, using the GNU C++ compiler version 4.3.2, under Linux (Fedora 10 distribution), with the compilation options `-O3 -DCGAL_NDEBUG`. The computer used was equipped with a 64bit Intel Xeon 3GHz processor and 32GB of RAM (a recent desktop machine as of 2009).

More benchmarks comparing CGAL to other software can be found in [LS05].

39.6.2 Memory Usage

We give here some indication about the memory usage of the triangulations. Those structures being intensively based on pointers, the size almost doubles on 64bit platforms compared to 32bit.

	Delaunay	Delaunay Fast location	Regular	Regular No hidden points
Construction from 10^2 points	0.00054	0.000576	0.000948	0.000955
Construction from 10^3 points	0.00724	0.00748	0.0114	0.0111
Construction from 10^4 points	0.0785	0.0838	0.122	0.117
Construction from 10^5 points	0.827	0.878	1.25	1.19
Construction from 10^6 points	8.5	9.07	12.6	12.2
Construction from 10^7 points	87.4	92.5	129	125
Point location in 10^2 points	9.93e-07	1.06e-06	7.19e-06	6.99e-06
Point location in 10^3 points	2.25e-06	1.93e-06	1.73e-05	1.76e-05
Point location in 10^4 points	4.79e-06	3.09e-06	3.96e-05	3.76e-05
Point location in 10^5 points	2.98e-05	6.12e-06	1.06e-04	1.06e-04
Point location in 10^6 points	1e-04	9.65e-06	2.7e-04	2.67e-04
Point location in 10^7 points	2.59e-04	1.33e-05	6.25e-04	6.25e-04
Vertex removal	1e-04	1.03e-04	1.42e-04	1.38e-04

Figure 39.6: Running times in seconds for algorithms on 3D triangulations.

The size also depends on the size of the point type which is copied in the vertices (hence on the kernel). Obviously, any user data added to vertices and cells also affect the memory used.

More specifically, the memory space used to store a triangulation is first a function of the size of its *Vertex* and *Cell* types times their numbers (and for volumic distribution, one sees about 6.7 times more cells than vertices). However, these are stored in memory using *Compact_container*, which allocates them in lists of blocks of growing size, and this requires some additional overhead for bookkeeping. Moreover, memory is only released to the system when clearing or destroying the triangulation. This can be important for algorithms like simplifications of data sets which will produce fragmented memory usage (doing fresh copies of the data structures are one way out in such cases). The asymptotic memory overhead of *Compact_container* for its internal bookkeeping is otherwise on the order of $O(\sqrt{n})$.

Figure 39.7 shows the number of bytes used per points, as measured empirically using *Memory_sizer* for large triangulations (10^6 random points).

	Delaunay	Delaunay Fast location	Regular	Regular No hidden points
32bit	274	291	336	282
64bit	519	553	635	527

Figure 39.7: Memory usage in bytes per point for large data sets.

39.6.3 Variability Depending on the Data Sets and the Kernel

Besides the complexity of the Delaunay triangulation that varies with the distribution of the points, another critical aspect affects the efficiency : the degeneracy of the data sets. These algorithms are quite sensitive to numerical accuracy and it is important to run them using exact predicates.

Using a kernel with no exact predicates will quickly lead to crashes or infinite loops once they are executed on non-random data sets. More precisely, problems appear with data sets which contain (nearly) degenerate cases for the *orientation* and *side_of_oriented_sphere* predicates, namely when there are (nearly) coplanar or (nearly) cospherical points. This unfortunately happens often in practice with data coming from various kinds of scanners or other automatic acquisition devices.

Using an inexact kernel such as *Simple_cartesian<double>* would lead to optimal performance, which is only about 30% better than *Exact_predicates_inexact_constructions_kernel*. The latter is strongly recommended since it takes care about potential robustness issues. The former can be used for benchmarking purposes mostly, or when you really know that your data sets won't exhibit any robustness issue.

Exact predicates take more time to compute when they hit (nearly) degenerate cases. Depending on the data set, this can have a visible impact on the overall performance of the algorithm or not.

Sometimes you need exact constructions as well, so *Exact_predicates_exact_constructions_kernel* is a must. This is the case for example when you need the *dual* functions to be exact, or when your input is stored in points of such a kernel for other reasons (because it is the output of another algorithm which has this requirement, for example). This will slow down the computations by a factor of 4 to 5 at least, and it can be much more.

Figure 39.8 gives more detailed timings about various kernels one the following data sets : random points in a cube, random points on the surface of an ellipsoid, points scanned on the surface of a Buddha statue, points on a molecular surface, and points scanned on a dryer handle. See Figure 39.9 for pictures of the last 3 objects, which respectively illustrate volumic data, surfacic data, and data with many degenerate cases. This last data set exhibits an infinite loop with an inexact kernel, and of course we are not sure whether what is computed for the other data sets with this inexact kernel is a Delaunay triangulation. General introductory information about these robustness issues can be found in [KMP+08]. More benchmarks around this issue can also be found in [DP03].

	Random	**Ellipsoid**	**Buddha**	**Molecule**	**Dryer Handle**
#points	**100000**	**100000**	**542548**	**525296**	**49787**
Simple_cartesian<double>	0.69	0.627	4.21	3.8	∞-loop
Exact_predicates_inexact_constructions_kernel	0.824	0.749	4.99	4.64	1.68
Exact_predicates_exact_constructions_kernel	4.59	3.85	30.1	26.4	4.57
Simple_cartesian<Gmpq>	492	534	1120	1030	75.2

Figure 39.8: Running times (seconds) for various kernels and data sets.

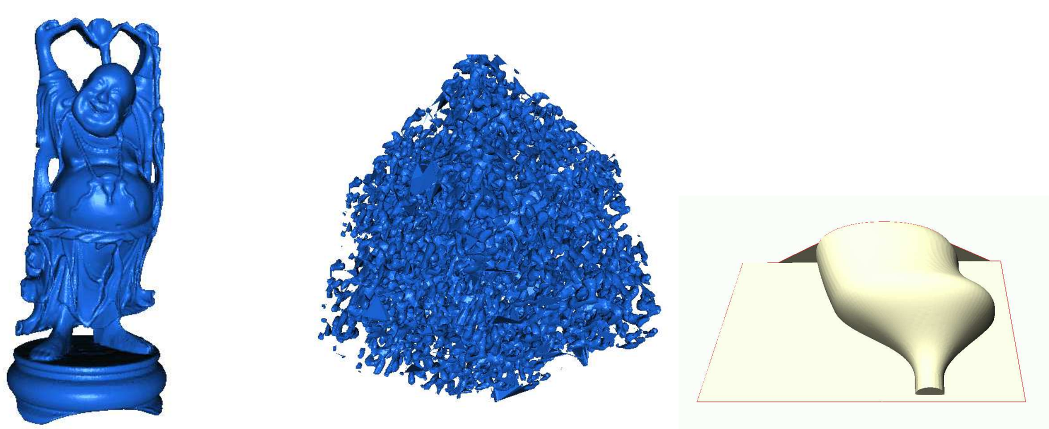


Figure 39.9: Data sets used in the benchmark of Figure 39.8.

39.7 Design and Implementation History

Monique Teillaud started to work on the 3D triangulation packages in 1997, following the design of the 2D triangulation packages. The notions of degenerate dimensions and infinite vertex were formalized [Tei99] and induced changes in the 2D triangulation packages. The packages were first released in CGAL 2.1. They contained basic functionalities on triangulations, Delaunay triangulations, regular triangulations.

A first version of removal of a vertex from a Delaunay triangulation was released in CGAL 2.2. However, this removal became really robust only in CGAL 2.3, after some research that allowed to deal with degenerate cases quite easily [DT03]. Andreas Fabri implemented this revised version of the removal, and a faster removal algorithm for CGAL 3.0.

The latter algorithm was proposed by Mariette Yvinec, who contributed in several ways to the package, first since she was maintaining the close 2D triangulation package and participated in many discussions, she also wrote the traits classes for regular triangulations.

In 2000, Sylvain Pion started working on these packages. He improved the efficiency of triangulations in CGAL 2.3 and 2.4 in several ways [BDP+02]: he implemented the Delaunay hierarchy [Dev02] in 2.3, he improved the memory footprint in 2.4 and 3.0, he also performed work on arithmetic filters [DP03] (see *Filtered_kernel*) to improve the speed of triangulations. He changed the design in CGAL 3.0, allowing users to add handles in their own vertices and cells.

Olivier Devillers, co-author of preliminary versions of the CGAL 2d triangulations, participated in many discussions, in particular about the perturbations, and more concretely in the implementation of the Delaunay hierarchy.

In 2005, Christophe Delage implemented the vertex removal function for regular triangulations, using the symbolic perturbation proposed in [DT06], which allowed to release this functionality in CGAL 3.2.

In 2006, Nico Kruithof wrote the *Triangulation_simplex_3* class that can store simplices of any dimension and improved the internal organization of the code.

As of March 2007, Christophe Delage made the iterator range insert methods and constructors use *spatial_sort* to improve efficiency.

In 2008, Camille Wormser added a few more iterators in the package that were integrated in release 3.4.

In 2009, Sylvain Pion simplified the design of the Delaunay hierarchy so that it became the simple *Fast_location* policy in release 3.6.

In 2010, Pedro de Castro and Olivier Devillers added the point displacement in release 3.7.

In 2011, Pedro de Castro and Olivier Devillers implemented in release 3.8 the structural filtering method, improving the efficiency of point location.

A new demo of this package was introduced in CGAL 3.8, coded by Fei (Sophie) Che, who was co-mentored by Manuel Caroli and Monique Teillaud in the framework of the Google Summer of Code, 2010.

The authors wish to thank Lutz Kettner for inspiring discussions about the design of CGAL. Jean-Daniel Boissonnat is also acknowledged [BDTY00].

3D Triangulations Reference Manual

Sylvain Pion and Monique Teillaud

A three-dimensional triangulation is a three-dimensional simplicial complex, pure connected and without singularities [BY98]. Its cells (3-faces) are such that two cells either do not intersect or share a common facet (2-face), edge (1-face) or vertex (0-face).

The basic 3D-triangulation class of CGAL is primarily designed to represent the triangulations of a set of points A in $\mathbb{R}^3$. It can be viewed as a partition of the convex hull of A into tetrahedra whose vertices are the points of A. Together with the unbounded cell having the convex hull boundary as its frontier, the triangulation forms a partition of $\mathbb{R}^3$.

In order to deal only with tetrahedra, which is convenient for many applications, the unbounded cell can be subdivided into tetrahedra by considering that each convex hull facet is incident to an *infinite cell* having as fourth vertex an auxiliary vertex called the *infinite vertex*. In that way, each facet is incident to exactly two cells and special cases at the boundary of the convex hull are simple to deal with.

A triangulation is a collection of vertices and cells that are linked together through incidence and adjacency relations. Each cell gives access to its four incident vertices and to its four adjacent cells. Each vertex gives access to one of its incident cells.

The four vertices of a cell are indexed with 0, 1, 2 and 3 in positive orientation, the positive orientation being defined by the orientation of the underlying Euclidean space $\mathbb{R}^3$. The neighbors of a cell are also indexed with 0, 1, 2, 3 in such a way that the neighbor indexed by i is opposite to the vertex with the same index. See Figure 39.1.

39.8 Classified Reference Pages

Concepts

Classes

Main Classes

Traits Classes

Enums

39.9 Alphabetical List of Reference Pages

Class

CGAL::Triangulation_3<TriangulationTraits_3,TriangulationDataStructure_3>

Definition

The class *Triangulation_3* represents a 3-dimensional tetrahedralization of points.

#include <CGAL/Triangulation_3.h>

Parameters

The first template argument must be a model of the *TriangulationTraits_3* concept.

The second template argument must be a model of the *TriangulationDataStructure_3* concept. It has the default value *Triangulation_data_structure_3< Triangulation_vertex_base_3<TriangulationTraits_3> ,Triangulation_cell_base_3<TriangulationTraits_3> >*.

Inherits From

Triangulation_utils_3

Types

The class *Triangulation_3* defines the following types:

typedef TriangulationDataStructure_3	*Triangulation_data_structure;*
typedef TriangulationTraits_3	*Geom_traits;*

typedef TriangulationTraits_3::Point_3	*Point;*
typedef TriangulationTraits_3::Segment_3	*Segment;*
typedef TriangulationTraits_3::Triangle_3	*Triangle;*
typedef TriangulationTraits_3::Tetrahedron_3	*Tetrahedron;*

Only vertices (0-faces) and cells (3-faces) are stored. Edges (1-faces) and facets (2-faces) are not explicitly represented and thus there are no corresponding classes (see Section 39.1).

typedef TriangulationDataStructure_3::Vertex	*Vertex;*
typedef TriangulationDataStructure_3::Cell	*Cell;*
typedef TriangulationDataStructure_3::Facet	*Facet;*
typedef TriangulationDataStructure_3::Edge	*Edge;*

The vertices and faces of the triangulations are accessed through *handles*, *iterators* and *circulators*. A handle is a model of the *Handle* concept, and supports the two dereference operators *operator** and *operator->*. A circulator is a model of the concept *Circulator*. Iterators and circulators are bidirectional and non-mutable. The edges and facets of the triangulation can also be visited through iterators and circulators which are bidirectional and non-mutable.

Iterators and circulators are convertible to the corresponding handles, thus the user can pass them directly as arguments to the functions.

typedef TriangulationDataStructure_3::Vertex_handle	*Vertex_handle;*	handle to a vertex
typedef TriangulationDataStructure_3::Cell_handle	*Cell_handle;*	handle to a cell
typedef Triangulation_simplex_3<Self>	*Simplex;*	Reference to a simplex (vertex, edge, facet or cell) of the triangulation
typedef TriangulationDataStructure_3::size_type	*size_type;*	Size type (an unsigned integral type)
typedef TriangulationDataStructure_3::difference_type	*difference_type;*	Difference type (a signed integral type)
typedef TriangulationDataStructure_3::Cell_iterator	*All_cells_iterator;*	iterator over cells
typedef TriangulationDataStructure_3::Facet_iterator	*All_facets_iterator;*	iterator over facets
typedef TriangulationDataStructure_3::Edge_iterator	*All_edges_iterator;*	iterator over edges
typedef TriangulationDataStructure_3::Vertex_iterator	*All_vertices_iterator;*	iterator over vertices

Triangulation_3<TriangulationTraits_3,TriangulationDataStructure_3>:: Finite_cells_iterator

iterator over finite cells

Triangulation_3<TriangulationTraits_3,TriangulationDataStructure_3>:: Finite_facets_iterator

iterator over finite facets

Triangulation_3<TriangulationTraits_3,TriangulationDataStructure_3>:: Finite_edges_iterator

iterator over finite edges

Triangulation_3<TriangulationTraits_3,TriangulationDataStructure_3>:: Finite_vertices_iterator

iterator over finite vertices

Triangulation_3<TriangulationTraits_3,TriangulationDataStructure_3>:: Point_iterator

iterator over the points corresponding to the finite vertices of the triangulation.

typedef TriangulationDataStructure_3::Cell_circulator	*Cell_circulator;*	circulator over all cells incident to a given edge
typedef TriangulationDataStructure_3::Facet_circulator	*Facet_circulator;*	circulator over all facets incident to a given edge

The triangulation class also defines the following enum type to specify which case occurs when locating a point in the triangulation.

enum Locate_type { *VERTEX=0,*
EDGE,
FACET,
CELL,
OUTSIDE_CONVEX_HULL,
OUTSIDE_AFFINE_HULL}

Creation

Triangulation_3<TriangulationTraits_3,TriangulationDataStructure_3> t(TriangulationTraits_3 traits = TriangulationTraits_3())

Introduces a triangulation *t* having only one vertex which is the infinite vertex.

Triangulation_3<TriangulationTraits_3,TriangulationDataStructure_3> t(Triangulation_3 tr);

Copy constructor. All vertices and faces are duplicated.

template < class InputIterator>
Triangulation_3<TriangulationTraits_3,TriangulationDataStructure_3> t(InputIterator first, InputIterator last, TriangulationTraits_3 traits = TriangulationTraits_3())

Equivalent to constructing an empty triangulation with the optional traits class argument and calling *insert(first,last)*.

Assignment

Triangulation_3 & *t = Triangulation_3 tr*

The triangulation *tr* is duplicated, and modifying the copy after the duplication does not modify the original. The previous triangulation held by *t* is deleted.

void *t.swap(Triangulation_3 & tr)*

The triangulations *tr* and *t* are swapped. *t.swap(tr)* should be preferred to *t* = *tr* or to *t(tr)* if *tr* is deleted after that. Indeed, there is no copy of cells and vertices, thus this method runs in constant time.

void *t.clear()* Deletes all finite vertices and all cells of *t*.

template < class GT, class Tds1, class Tds2 >

bool *Triangulation_3<GT, Tds1> t1* == *Triangulation_3<GT, Tds2> t2*

Equality operator. Returns true iff there exist a bijection between the vertices of *t1* and those of *t2* and a bijection between the cells of *t1* and those of *t2*, which preserve the geometry of the triangulation, that is, the points of each corresponding pair of vertices are equal, and the tetrahedra corresponding to each pair of cells are equal (up to a permutation of their vertices).

template < class GT, class Tds1, class Tds2 >
bool *Triangulation_3<GT, Tds1> t1* != *Triangulation_3<GT, Tds2> t2*

The opposite of *operator*==.

Access Functions

TriangulationTraits_3	*t.geom_traits() const*	Returns a const reference to the geometric traits object.
TriangulationDataStructure_3	*t.tds() const*	Returns a const reference to the triangulation data structure.

advanced

Non const access

The responsibility of keeping a valid triangulation belongs to the user when using advanced operations allowing a direct manipulation of the *tds*.

TriangulationDataStructure_3&	*t.tds()*	Returns a reference to the triangulation data structure.

This method is mainly a help for users implementing their own triangulation algorithms.

advanced

int	*t.dimension() const*	Returns the dimension of the affine hull.
size_type	*t.number_of_vertices() const*	Returns the number of finite vertices.
size_type	*t.number_of_cells() const*	Returns the number of cells or 0 if *t.dimension()*< 3.
Vertex_handle	*t.infinite_vertex()*	Returns the infinite vertex.
Cell_handle	*t.infinite_cell() const*	Returns a cell incident to the infinite vertex.

Non-constant-time access functions

As previously said, the triangulation is a collection of cells that are either infinite or represent a finite tetrahedra, where an infinite cell is a cell incident to the infinite vertex. Similarly we call an edge (resp. facet) *infinite* if it is incident to the infinite vertex.

size_type	*t.number_of_facets() const*	The number of facets. Returns 0 if *t.dimension()*<2.
size_type	*t.number_of_edges() const*	The number of edges. Returns 0 if *t.dimension()*<1.

size_type *t.number_of_finite_cells() const*

The number of finite cells. Returns 0 if *t.dimension()*<3.

size_type *t.number_of_finite_facets() const*

The number of finite facets. Returns 0 if *t.dimension()*<2.

size_type *t.number_of_finite_edges() const*

The number of finite edges. Returns 0 if *t.dimension()*<1.

Geometric access functions

Tetrahedron *t.tetrahedron(Cell_handle c) const*

Returns the tetrahedron formed by the four vertices of c.
Precondition: *t.dimension()* $=3$ and the cell is finite.

Triangle *t.triangle(Cell_handle c, int i) const*

Returns the triangle formed by the three vertices of facet *(c,i)*. The triangle is oriented so that its normal points to the inside of cell c.
Precondition: *t.dimension()* ≥ 2 and $i \in \{0,1,2,3\}$ in dimension 3, $i=3$ in dimension 2, and the facet is finite.

Triangle *t.triangle(Facet f) const*

Same as the previous method for facet f.
Precondition: *t.dimension()* ≥ 2 and the facet is finite.

Segment *t.segment(Edge e) const*

Returns the line segment formed by the vertices of e.
Precondition: *t.dimension()* ≥ 1 and e is finite.

Segment *t.segment(Cell_handle c, int i, int j) const*

Same as the previous method for edge *(c,i,j)*.
Precondition: As above and $i \neq j$. Moreover $i,j \in \{0,1,2,3\}$ in dimension 3, $i,j \in \{0,1,2\}$ in dimension 2, $i,j \in \{0,1\}$ in dimension 1, and the edge is finite.

Point *t.point(Cell_handle c, int i) const*

Returns the point given by vertex i of cell c.
Precondition: *t.dimension()* ≥ 0 and $i \in \{0,1,2,3\}$ in dimension 3, $i \in \{0,1,2\}$ in dimension 2, $i \in \{0,1\}$ in dimension 1, $i=0$ in dimension 0, and the vertex is finite.

Point *t.point(Vertex_handle v) const*

Same as the previous method for vertex v.
Precondition: *t.dimension()* ≥ 0 and the vertex is finite.

Tests for Finite and Infinite Vertices and Faces

bool *t.is_infinite(Vertex_handle v) const*

true, iff vertex v is the infinite vertex.

bool *t.is_infinite(Cell_handle c) const*

true, iff c is incident to the infinite vertex.
Precondition: *t.dimension()* $= 3$.

bool *t.is_infinite(Cell_handle c, int i) const*

true, iff the facet i of cell c is incident to the infinite vertex.
Precondition: *t.dimension()* ≥ 2 and $i \in \{0,1,2,3\}$ in dimension 3, $i = 3$ in dimension 2.

bool *t.is_infinite(Facet f) const*

true iff facet f is incident to the infinite vertex.
Precondition: *t.dimension()* ≥ 2.

bool *t.is_infinite(Cell_handle c, int i, int j) const*

true, iff the edge *(i,j)* of cell c is incident to the infinite vertex.
Precondition: *t.dimension()* ≥ 1 and $i \neq j$. Moreover $i, j \in \{0,1,2,3\}$ in dimension 3, $i, j \in \{0,1,2\}$ in dimension 2, $i, j \in \{0,1\}$ in dimension 1.

bool *t.is_infinite(Edge e) const*

true iff edge e is incident to the infinite vertex.
Precondition: *t.dimension()* ≥ 1.

Queries

bool *t.is_vertex(Point p, Vertex_handle & v) const*

Tests whether p is a vertex of t by locating p in the triangulation. If p is found, the associated vertex v is given.

bool *t.is_vertex(Vertex_handle v) const*

Tests whether v is a vertex of t.

bool *t.is_edge(Vertex_handle u, Vertex_handle v, Cell_handle & c, int & i, int & j) const*

Tests whether *(u,v)* is an edge of t. If the edge is found, it gives a cell c having this edge and the indices i and j of the vertices u and v in c, in this order.
Precondition: u and v are vertices of t.

bool *t.is_facet(Vertex_handle u,*
Vertex_handle v,
Vertex_handle w,
Cell_handle & c,
int & i,
int & j,

int & k) const

Tests whether *(u,v,w)* is a facet of *t*. If the facet is found, it computes a cell *c* having this facet and the indices *i*, *j* and *k* of the vertices *u*, *v* and *w* in *c*, in this order.
Precondition: *u*, *v* and *w* are vertices of *t*.

bool *t.is_cell(Cell_handle c) const*

Tests whether *c* is a cell of *t*.

bool *t.is_cell(Vertex_handle u,*
Vertex_handle v,
Vertex_handle w,
Vertex_handle x,
Cell_handle & c,
int & i,
int & j,
int & k,
int & l) const

Tests whether *(u,v,w,x)* is a cell of *t*. If the cell *c* is found, the method computes the indices *i*, *j*, *k* and *l* of the vertices *u*, *v*, *w* and *x* in *c*, in this order.
Precondition: *u*, *v*, *w* and *x* are vertices of *t*.

bool *t.is_cell(Vertex_handle u,*
Vertex_handle v,
Vertex_handle w,
Vertex_handle x,
Cell_handle & c) const

Tests whether *(u,v,w,x)* is a cell of *t* and computes this cell *c*.
Precondition: *u*, *v*, *w* and *x* are vertices of *t*.

There is a method *has_vertex* in the cell class. The analogous methods for facets are defined here.

bool *t.has_vertex(Facet f, Vertex_handle v, int & j) const*

If *v* is a vertex of *f*, then *j* is the index of *v* in the cell *f.first*, and the method returns *true*.
Precondition: *t*.dimension()=3

bool *t.has_vertex(Cell_handle c, int i, Vertex_handle v, int & j) const*

Same for facet *(c,i)*. Computes the index *j* of *v* in *c*.

bool *t.has_vertex(Facet f, Vertex_handle v) const*
bool *t.has_vertex(Cell_handle c, int i, Vertex_handle v) const*

Same as the first two methods, but these two methods do not return the index of the vertex.

The following three methods test whether two facets have the same vertices.

bool *t.are_equal(Cell_handle c, int i, Cell_handle n, int j) const*
bool *t.are_equal(Facet f, Facet g) const*

bool *t.are_equal(Facet f, Cell_handle n, int j) const*

For these three methods:
Precondition: t.dimension()=3.

Point location

The class *Triangulation_3<TriangulationTraits_3,TriangulationDataStructure_3>* provides two functions to locate a given point with respect to a triangulation. It provides also functions to test if a given point is inside a finite face or not. Note that the class *Delaunay_triangulation_3* also provides a *nearest_vertex()* function.

Cell_handle *t.locate(Point query, Cell_handle start = Cell_handle())*

If the point *query* lies inside the convex hull of the points, the cell that contains the query in its interior is returned. If *query* lies on a facet, an edge or on a vertex, one of the cells having *query* on its boundary is returned.
If the point *query* lies outside the convex hull of the points, an infinite cell with vertices $\{p,q,r,\infty\}$ is returned such that the tetrahedron $(p,q,r,query)$ is positively oriented (the rest of the triangulation lies on the other side of facet (p,q,r)).
Note that locate works even in degenerate dimensions: in dimension 2 (resp. 1, 0) the *Cell_handle* returned is the one that represents the facet (resp. edge, vertex) containing the query point.
The optional argument *start* is used as a starting place for the search.

Cell_handle *t.locate(Point query, Vertex_handle hint) const*

Same as above but uses *hint* as the starting place for the search.

Cell_handle *t.locate(Point query, Locate_type & lt, int & li, int & lj, Cell_handle start = Cell_handle())*

If *query* lies inside the affine hull of the points, the *k*-face (finite or infinite) that contains *query* in its interior is returned, by means of the cell returned together with *lt*, which is set to the locate type of the query (*VERTEX, EDGE, FACET, CELL*, or *OUTSIDE_CONVEX_HULL* if the cell is infinite and *query* lies strictly in it) and two indices *li* and *lj* that specify the *k*-face of the cell containing *query*.
If the *k*-face is a cell, *li* and *lj* have no meaning; if it is a facet (resp. vertex), *li* gives the index of the facet (resp. vertex) and *lj* has no meaning; if it is and edge, *li* and *lj* give the indices of its vertices.
If the point *query* lies outside the affine hull of the points, which can happen in case of degenerate dimensions, *lt* is set to *OUTSIDE_AFFINE_HULL*, and the cell returned has no meaning. As a particular case, if there is no finite vertex yet in the triangulation, *lt* is set to *OUTSIDE_AFFINE_HULL* and *locate* returns the default constructed handle.
The optional argument *start* is used as a starting place for the search.

Cell_handle *t.locate(Point query, Locate_type & lt, int & li, int & lj, Vertex_handle hint) const*

Same as above but uses *hint* as the starting place for the search.

Bounded_side　*t.side_of_cell(Point p, Cell_handle c, Locate_type & lt, int & li, int & lj) const*

Returns a value indicating on which side of the oriented boundary of c the point p lies. More precisely, it returns:
- *ON_BOUNDED_SIDE* if p is inside the cell. For an infinite cell this means that p lies strictly in the half space limited by its finite facet and not containing any other point of the triangulation.
- *ON_BOUNDARY* if p on the boundary of the cell. For an infinite cell this means that p lies on the *finite* facet. Then *lt* together with *li* and *lj* give the precise location on the boundary. (See the descriptions of the *locate* methods.)
- *ON_UNBOUNDED_SIDE* if p lies outside the cell. For an infinite cell this means that p does not satisfy either of the two previous conditions.
Precondition: *t.dimension()* $= 3$

Bounded_side　*t.side_of_facet(Point p, Facet f, Locate_type & lt, int & li, int & lj) const*

Returns a value indicating on which side of the oriented boundary of f the point p lies:
- *ON_BOUNDED_SIDE* if p is inside the facet. For an infinite facet this means that p lies strictly in the half plane limited by its finite edge and not containing any other point of the triangulation .
- *ON_BOUNDARY* if p is on the boundary of the facet. For an infinite facet this means that p lies on the finite edge. *lt*, *li* and *lj* give the precise location of p on the boundary of the facet. *li* and *lj* refer to indices in the degenerate cell c representing f.
- *ON_UNBOUNDED_SIDE* if p lies outside the facet. For an infinite facet this means that p does not satisfy either of the two previous conditions.

Precondition: *t.dimension()* $= 2$ and p lies in the plane containing the triangulation. *f.second* $= 3$ (in dimension 2 there is only one facet per cell).

Bounded_side　*t.side_of_facet(Point p, Cell_handle c, Locate_type & lt, int & li, int & lj) const*

Same as the previous method for the facet *(c,3)*.

Bounded_side　*t.side_of_edge(Point p, Edge e, Locate_type & lt, int & li) const*

Returns a value indicating on which side of the oriented boundary of e the point p lies:
- *ON_BOUNDED_SIDE* if p is inside the edge. For an infinite edge this means that p lies in the half line defined by the vertex and not containing any other point of the triangulation.
- *ON_BOUNDARY* if p equals one of the vertices, *li* give the index of the vertex in the cell storing e
- *ON_UNBOUNDED_SIDE* if p lies outside the edge. For an infinite edge this means that p lies on the other half line, which contains the other points of the triangulation.
Precondition: *t.dimension()* $= 1$ and p is collinear with the points of the triangulation. *e.second* $= 0$ and *e.third* $= 1$ (in dimension 1 there is only one edge per cell).

Bounded_side　*t.side_of_edge(Point p, Cell_handle c, Locate_type & lt, int & li) const*

Same as the previous method for edge $(c,0,1)$.

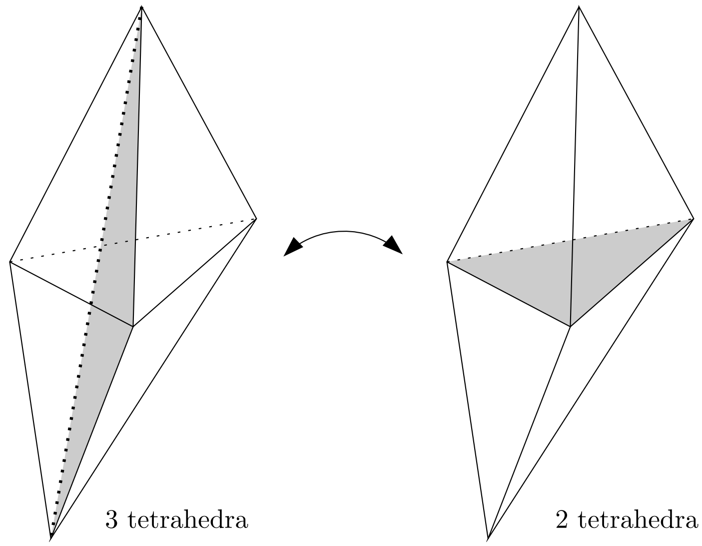


Figure 39.10: Flips.

Flips

Two kinds of flips exist for a three-dimensional triangulation. They are reciprocal. To be flipped, an edge must be incident to three tetrahedra. During the flip, these three tetrahedra disappear and two tetrahedra appear. Figure 39.10(left) shows the edge that is flipped as bold dashed, and one of its three incident facets is shaded. On the right, the facet shared by the two new tetrahedra is shaded.

Flips are possible only under the following conditions:
- the edge or facet to be flipped is not on the boundary of the convex hull of the triangulation
- the five points involved are in convex position.

The following methods guarantee the validity of the resulting 3D triangulation.

Flips for a 2d triangulation are not implemented yet

bool	*t.flip(Edge e)*
bool	*t.flip(Cell_handle c, int i, int j)*

Before flipping, these methods check that edge *e=(c,i,j)* is flippable (which is quite expensive). They return *false* or *true* according to this test.

void	*t.flip_flippable(Edge e)*
void	*t.flip_flippable(Cell_handle c, int i, int j)*

Should be preferred to the previous methods when the edge is known to be flippable.
Precondition: The edge is flippable.

bool	*t.flip(Facet f)*
bool	*t.flip(Cell_handle c, int i)*

Before flipping, these methods check that facet *f=(c,i)* is flippable (which is quite expensive). They return *false* or *true* according to this test.

void *t.flip_flippable(Facet f)*
void *t.flip_flippable(Cell_handle c, int i)*

Should be preferred to the previous methods when the facet is known to be flippable.
Precondition: The facet is flippable.

Insertions

The following operations are guaranteed to lead to a valid triangulation when they are applied on a valid triangulation.

Vertex_handle *t.insert(Point p, Cell_handle start = Cell_handle())*

Inserts point *p* in the triangulation and returns the corresponding vertex.
If point *p* coincides with an already existing vertex, this vertex is returned and the triangulation remains unchanged.
If point *p* lies in the convex hull of the points, it is added naturally: if it lies inside a cell, the cell is split into four cells, if it lies on a facet, the two incident cells are split into three cells, if it lies on an edge, all the cells incident to this edge are split into two cells.
If point *p* is strictly outside the convex hull but in the affine hull, *p* is linked to all visible points on the convex hull to form the new triangulation. See Figure 39.11.
If point *p* is outside the affine hull of the points, *p* is linked to all the points, and the dimension of the triangulation is incremented. All the points now belong to the boundary of the convex hull, so, the infinite vertex is linked to all the points to triangulate the new infinite face. See Figure 39.12. The optional argument *start* is used as a starting place for the search.

Vertex_handle *t.insert(Point p, Vertex_handle hint)*

Same as above but uses *hint* as the starting place for the search.

Vertex_handle *t.insert(Point p, Locate_type lt, Cell_handle loc, int li, int lj)*

Inserts point *p* in the triangulation and returns the corresponding vertex. Similar to the above *insert()* function, but takes as additional parameter the return values of a previous location query. See description of *locate()* above.

template < class InputIterator >
std::ptrdiff_t *t.insert(InputIterator first, InputIterator last)*

Inserts the points in the range $[first, last)$. Returns the number of inserted points. Note that this function is not guaranteed to insert the points following the order of *InputIterator*.
Precondition: The *value_type* of *first* and *last* is *Point*.

The previous methods are sufficient to build a whole triangulation. We also provide some other methods that can be used instead of *insert(p)* when the place where the new point *p* must be inserted is already known. They are also guaranteed to lead to a valid triangulation when they are applied on a valid triangulation.

Vertex_handle *t.insert_in_cell(Point p, Cell_handle c)*

Inserts point p in cell c. Cell c is split into 4 tetrahedra.
Precondition: *t.dimension()* $= 3$ and p lies strictly inside cell c.

Vertex_handle *t.insert_in_facet(Point p, Facet f)*

Inserts point p in facet f. In dimension 3, the 2 neighboring cells are split into 3 tetrahedra; in dimension 2, the facet is split into 3 triangles.
Precondition: *t.dimension()* ≥ 2 and p lies strictly inside face f.

Vertex_handle *t.insert_in_facet(Point p, Cell_handle c, int i)*

As above, insertion in facet *(c,i)*.
Precondition: As above and $i \in \{0,1,2,3\}$ in dimension 3, $i = 3$ in dimension 2.

Vertex_handle *t.insert_in_edge(Point p, Edge e)*

Inserts p in edge e. In dimension 3, all the cells having this edge are split into 2 tetrahedra; in dimension 2, the 2 neighboring facets are split into 2 triangles; in dimension 1, the edge is split into 2 edges.
Precondition: *t.dimension()* ≥ 1 and p lies on edge e.

Vertex_handle *t.insert_in_edge(Point p, Cell_handle c, int i, int j)*

As above, inserts p in edge (i,j) of c.
Precondition: As above and $i \neq j$. Moreover $i, j \in \{0,1,2,3\}$ in dimension 3, $i, j \in \{0,1,2\}$ in dimension 2, $i, j \in \{0,1\}$ in dimension 1.

Vertex_handle *t.insert_outside_convex_hull(Point p, Cell_handle c)*

The cell c must be an infinite cell containing p.
Links p to all points in the triangulation that are visible from p. Updates consequently the infinite faces. See Figure 39.11.
Precondition: *t.dimension()* > 0, c, and the k-face represented by c is infinite and contains t.

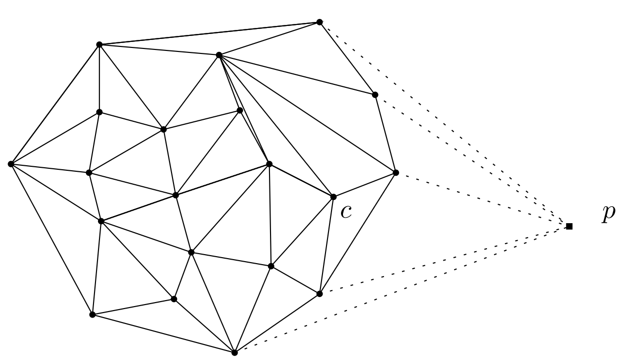


Figure 39.11: *insert_outside_convex_hull* (2-dimensional case).

Vertex_handle   *t.insert_outside_affine_hull(Point p)*

p is linked to all the points, and the infinite vertex is linked to all the points (including p) to triangulate the new infinite face, so that all the points now belong to the boundary of the convex hull. See Figure 39.12.
This method can be used to insert the first point in an empty triangulation.
Precondition: *t.dimension()* < 3 and p lies outside the affine hull of the points.

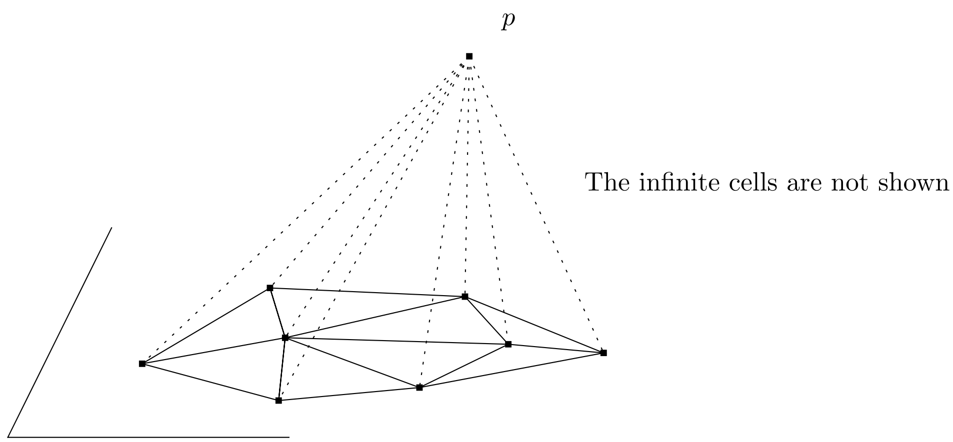


Figure 39.12: *insert_outside_affine_hull* (2-dimensional case).

template <class CellIt>
Vertex_handle   *t.insert_in_hole(Point p, CellIt cell_begin, CellIt cell_end, Cell_handle begin, int i)*

Creates a new vertex by starring a hole. It takes an iterator range [*cell_begin*; *cell_end*[of *Cell_handle*s which specifies a hole: a set of connected cells (resp. facets in dimension 2) which is star-shaped wrt p. (*begin*, i) is a facet (resp. an edge) on the boundary of the hole, that is, *begin* belongs to the set of cells (resp. facets) previously described, and *begin->neighbor(i)* does not. Then this function deletes all the cells (resp. facets) describing the hole, creates a new vertex v, and for each facet (resp. edge) on the boundary of the hole, creates a new cell (resp. facet) with v as vertex. Then *v->set_point(p)* is called and v is returned.
This operation is equivalent to calling *tds().insert_in_hole(cell_begin, cell_end, begin, i); v->set_point(p)*.
Precondition: *t.dimension()* ≥ 2, the set of cells (resp. facets in dimension 2) is connected, its boundary is connected, and p lies inside the hole, which is star-shaped wrt p.

template <class CellIt>
Vertex_handle   *t.insert_in_hole(Point p,*
CellIt cell_begin,
CellIt cell_end,
Cell_handle begin,
int i,
Vertex_handle newv)

Same as above, except that *newv* will be used as the new vertex, which must have been allocated previously with e.g. *create_vertex*.

Traversal of the Triangulation

The triangulation class provides several iterators and circulators that allow one to traverse it (completely or partially).

Cell, Face, Edge and Vertex Iterators

The following iterators allow the user to visit cells, facets, edges and vertices of the triangulation. These iterators are non-mutable, bidirectional and their value types are respectively *Cell*, *Facet*, *Edge* and *Vertex*. They are all invalidated by any change in the triangulation.

Finite_vertices_iterator *t.finite_vertices_begin() const*

Starts at an arbitrary finite vertex. Then ++ and -- will iterate over finite vertices. Returns *finite_vertices_end()* when *t.number_of_vertices()* $= 0$.

Finite_vertices_iterator *t.finite_vertices_end() const*

Past-the-end iterator

Finite_edges_iterator *t.finite_edges_begin() const*

Starts at an arbitrary finite edge. Then ++ and -- will iterate over finite edges. Returns *finite_edges_end()* when *t.dimension()* < 1.

Finite_edges_iterator *t.finite_edges_end() const*

Past-the-end iterator

Finite_facets_iterator *t.finite_facets_begin() const*

Starts at an arbitrary finite facet. Then ++ and -- will iterate over finite facets. Returns *finite_facets_end()* when *t.dimension()* < 2.

Finite_facets_iterator *t.finite_facets_end() const*

Past-the-end iterator

Finite_cells_iterator *t.finite_cells_begin() const*

Starts at an arbitrary finite cell. Then ++ and -- will iterate over finite cells. Returns *finite_cells_end()* when *t.dimension()* < 3.

Finite_cells_iterator *t.finite_cells_end() const* Past-the-end iterator

All_vertices_iterator *t.all_vertices_begin() const*

Starts at an arbitrary vertex. Iterates over all vertices (even the infinite one). Returns *vertices_end()* when *t.number_of_vertices()* $= 0$.

All_vertices_iterator *t.all_vertices_end() const*

Past-the-end iterator

All_edges_iterator	*t.all_edges_begin() const*	
		Starts at an arbitrary edge. Iterates over all edges (even infinite ones). Returns *edges_end()* when *t.dimension()* < 1.
All_edges_iterator	*t.all_edges_end() const*	Past-the-end iterator
All_facets_iterator	*t.all_facets_begin() const*	
		Starts at an arbitrary facet. Iterates over all facets (even infinite ones). Returns *facets_end()* when *t.dimension()* < 2.
All_facets_iterator	*t.all_facets_end() const*	Past-the-end iterator
All_cells_iterator	*t.all_cells_begin() const*	Starts at an arbitrary cell. Iterates over all cells (even infinite ones). Returns *cells_end()* when *t.dimension()* < 3.
All_cells_iterator	*t.all_cells_end() const*	Past-the-end iterator
Point_iterator	*t.points_begin() const*	Iterates over the points of the triangulation.
Point_iterator	*t.points_end() const*	Past-the-end iterator

Cell and Facet Circulators

The following circulators respectively visit all cells or all facets incident to a given edge. They are non-mutable and bidirectional. They are invalidated by any modification of one of the cells traversed.

Cell_circulator *t.incident_cells(Edge e) const*

Starts at an arbitrary cell incident to *e*.
Precondition: *t.dimension()* $= 3$.

Cell_circulator *t.incident_cells(Cell_handle c, int i, int j) const*

As above for edge *(i,j)* of *c*.

Cell_circulator *t.incident_cells(Edge e, Cell_handle start) const*

Starts at cell *start*.
Precondition: *t.dimension()* $= 3$ and *start* is incident to *e*.

Cell_circulator *t.incident_cells(Cell_handle c, int i, int j, Cell_handle start) const*

As above for edge *(i,j)* of *c*.

The following circulators on facets are defined only in dimension 3, though facets are defined also in dimension 2: there are only two facets sharing an edge in dimension 2.

Facet_circulator *t.incident_facets(Edge e) const*

Starts at an arbitrary facet incident to *e*.
Precondition: *t.dimension()* $= 3$

Facet_circulator *t.incident_facets(Cell_handle c, int i, int j) const*

As above for edge *(i,j)* of *c*.

Facet_circulator *t.incident_facets(Edge e, Facet start) const*

Starts at facet *start*.
Precondition: *start* is incident to *e*.

Facet_circulator *t.incident_facets(Edge e, Cell_handle start, int f) const*

Starts at facet of index *f* in *start*.

Facet_circulator *t.incident_facets(Cell_handle c, int i, int j, Facet start) const*

As above for edge *(i,j)* of *c*.

Facet_circulator *t.incident_facets(Cell_handle c, int i, int j, Cell_handle start, int f) const*

As above for edge *(i,j)* of *c* and facet *(start,f)*.

Traversal of the incident cells, facets and edges, and the adjacent vertices of a given vertex

template <class OutputIterator>
OutputIterator *t.incident_cells(Vertex_handle v, OutputIterator cells) const*

Copies the *Cell_handle*s of all cells incident to *v* to the output iterator *cells*. Returns the resulting output iterator.
Precondition: $t.dimension() = 3$, $v \neq Vertex\_handle()$, *t.is_vertex(v)*.

template <class OutputIterator>
OutputIterator *t.finite_incident_cells(Vertex_handle v, OutputIterator cells) const*

Copies the *Cell_handle*s of all finite cells incident to *v* to the output iterator *cells*. Returns the resulting output iterator.
Precondition: $t.dimension() = 3$, $v \neq Vertex\_handle()$, *t.is_vertex(v)*.

template <class OutputIterator>
OutputIterator *t.incident_facets(Vertex_handle v, OutputIterator facets) const*

Copies all *Facet*s incident to *v* to the output iterator *facets*. Returns the resulting output iterator.
Precondition: $t.dimension() > 1$, $v \neq Vertex\_handle()$, *t.is_vertex(v)*.

template <class OutputIterator>
OutputIterator *t.finite_incident_facets(Vertex_handle v, OutputIterator facets) const*

Copies all finite *Facet*s incident to *v* to the output iterator *facets*. Returns the resulting output iterator.
Precondition: $t.dimension() > 1$, $v \neq Vertex\_handle()$, *t.is_vertex(v)*.

template <class OutputIterator>
OutputIterator *t.incident_edges(Vertex_handle v, OutputIterator edges) const*

Copies all *Edge*s incident to *v* to the output iterator *edges*. Returns the resulting output iterator.
Precondition: $t.dimension() > 0$, $v \neq Vertex\_handle()$, *t.is_vertex(v)*.

template <class OutputIterator>
OutputIterator *t.finite_incident_edges(Vertex_handle v, OutputIterator edges) const*

Copies all finite *Edge*s incident to *v* to the output iterator *edges*. Returns the resulting output iterator.
Precondition: *t.dimension()* > 0, $v \neq$ *Vertex_handle()*, *t.is_vertex(v)*.

template <class OutputIterator>
OutputIterator *t.adjacent_vertices(Vertex_handle v, OutputIterator vertices) const*

Copies the *Vertex_handle*s of all vertices adjacent to *v* to the output iterator *vertices*. If *t.dimension()* < 0, then do nothing. Returns the resulting output iterator.
Precondition: $v \neq$ *Vertex_handle()*, *t.is_vertex(v)*.

template <class OutputIterator>
OutputIterator *t.finite_adjacent_vertices(Vertex_handle v, OutputIterator vertices) const*

Copies the *Vertex_handle*s of all finite vertices adjacent to *v* to the output iterator *vertices*. If *t.dimension()* < 0, then do nothing. Returns the resulting output iterator.
Precondition: $v \neq$ *Vertex_handle()*, *t.is_vertex(v)*.

size_type *t.degree(Vertex_handle v) const*

Returns the degree of a vertex, that is, the number of incident vertices. The infinite vertex is counted.
Precondition: $v \neq$ *Vertex_handle()*, *t.is_vertex(v)*.

Traversal between adjacent cells

int *t.mirror_index(Cell_handle c, int i) const*

Returns the index of *c* in its i^{th} neighbor.
Precondition: $i \in \{0,1,2,3\}$.

Vertex_handle *t.mirror_vertex(Cell_handle c, int i) const*

Returns the vertex of the i^{th} neighbor of *c* that is opposite to *c*.
Precondition: $i \in \{0,1,2,3\}$.

Facet *t.mirror_facet(Facet f) const* Returns the same facet seen from the other adjacent cell.

debugging support

Checking

The responsibility of keeping a valid triangulation belongs to the user when using advanced operations allowing a direct manipulation of cells and vertices. We provide the user with the following methods to help debugging.

bool *t.is_valid(bool verbose = false) const*

Checks the combinatorial validity of the triangulation. Checks also the validity of its geometric embedding (see Section 39.1).
When *verbose* is set to true, messages describing the first invalidity encountered are printed.

bool *t.is_valid(Cell_handle c, bool verbose = false) const*

Checks the combinatorial validity of the cell by calling the *is_valid* method of the *TriangulationDataStructure_3* cell class. Also checks the geometric validity of *c*, if *c* is finite. (See Section 2610.)
When *verbose* is set to *true*, messages are printed to give a precise indication of the kind of invalidity encountered.

debugging support

I/O

CGAL provides an interface to Geomview for a 3D-triangulation, see Chapter 80 on *Geomview_stream*. *#include <CGAL/IO/Triangulation_geomview_ostream_3.h>*

istream&	*istream& is >> Triangulation_3 &t*	Reads the underlying combinatorial triangulation from *is* by calling the corresponding input operator of the triangulation data structure class (note that the infinite vertex is numbered 0), and the non-combinatorial information by calling the corresponding input operators of the vertex and the cell classes (such as point coordinates), which are provided by overloading the stream operators of the vertex and cell types. Assigns the resulting triangulation to *t*.
ostream&	*ostream& os << Triangulation_3 t*	Writes the triangulation *t* into *os*.

The information in the *iostream* is: the dimension, the number of finite vertices, the non-combinatorial information about vertices (point, etc; note that the infinite vertex is numbered 0), the number of cells, the indices of the vertices of each cell, plus the non-combinatorial information about each cell, then the indices of the neighbors of each cell, where the index corresponds to the preceding list of cells. When dimension < 3, the same information is stored for faces of maximal dimension instead of cells.

See Also

TriangulationDataStructure_3::Vertex
TriangulationDataStructure_3::Cell

Class

CGAL::Delaunay_triangulation_3<DelaunayTriangulationTraits_3,TriangulationDataStructure_3,LocationPolicy>

Definition

The class *Delaunay_triangulation_3* represents a three-dimensional Delaunay triangulation.

#include <CGAL/Delaunay_triangulation_3.h>

Parameters

The first template argument must be a model of the *DelaunayTriangulationTraits_3* concept.

The second template argument must be a model of the *TriangulationDataStructure_3* concept. It has the default value *Triangulation_data_structure_3<Triangulation_vertex_base_3<DelaunayTriangulationTraits_3>, Triangulation_cell_base_3<DelaunayTriangulationTraits_3> >*.

The third template argument is a tag which must be a *Location_policy<Tag>* : either *Fast_location* or *Compact_location*. *Fast_location* offers faster ($O(\log n)$ time) point location, which can be beneficial when performing point locations or random point insertions (with no good location hint) in large data sets. It is currently implemented using an additional triangulation hierarchy data structure [Dev02]. The default is *Compact_location*, which saves memory (3-5%) by avoiding the need for this separate data structure, and point location is then performed roughly in $O(n^{1/3})$ time. Note that this argument can also come in second position, which can be useful when the default value for the *TriangulationDataStructure_3* parameter is satisfactory (this is using so-called deduced parameters). Note that this argument replaces the functionality provided before CGAL 3.6 by *Triangulation_hierarchy_3*. An example of use can be found in the user manual 39.5.5.

Inherits From

Triangulation_3<DelaunayTriangulationTraits_3,Delaunay_triangulation_3<DelaunayTriangulationTraits_3,TriangulationDataStructure_3,LocationPolicy>::Triangulation_data_structure >

Types

typedef LocationPolicy *Location_policy;*

In addition to those inherited, the following types are defined, for use by the construction of the Voronoi diagram:

typedef DelaunayTriangulationTraits_3::Line_3 *Line;*
typedef DelaunayTriangulationTraits_3::Ray_3 *Ray;*
typedef DelaunayTriangulationTraits_3::Plane_3 *Plane;*
typedef DelaunayTriangulationTraits_3::Object_3 *Object;*

Creation

Delaunay_triangulation_3<DelaunayTriangulationTraits_3,TriangulationDataStructure_3,LocationPolicy> dt(DelaunayTriangulationTraits_3 traits = DelaunayTriangulationTraits_3())

Creates an empty Delaunay triangulation, possibly specifying a traits class *traits*.

Delaunay_triangulation_3<DelaunayTriangulationTraits_3,TriangulationDataStructure_3,LocationPolicy>
dt(Delaunay_triangulation_3 dt1)

Copy constructor.

template < class InputIterator >
Delaunay_triangulation_3<DelaunayTriangulationTraits_3,TriangulationDataStructure_3,LocationPolicy>
dt(InputIterator first,

In-

putIterator last,

DelaunayTriangulati

3 traits = DelaunayTriangulationTraits_3())

Equivalent to constructing an empty triangulation with the optional traits class argument and calling *insert(first,last)*.

Operations

Insertion

The following methods overload the corresponding methods of triangulations to ensure the empty sphere property of Delaunay triangulations.

In the degenerate case when there are co-spherical points, the Delaunay triangulation is known not to be uniquely defined. In this case, CGAL chooses a particular Delaunay triangulation using a symbolic perturbation scheme [DT03].

Vertex_handle *dt.insert(Point p, Cell_handle start = Cell_handle())*

Inserts point *p* in the triangulation and returns the corresponding vertex. Similar to the insertion in a triangulation, but ensures in addition the empty sphere property of all the created faces. The optional argument *start* is used as a starting place for the search.

Vertex_handle *dt.insert(Point p, Vertex_handle hint)*

Same as above but uses *hint* as a starting place for the search.

Vertex_handle *dt.insert(Point p, Locate_type lt, Cell_handle loc, int li, int lj)*

Inserts point *p* in the triangulation and returns the corresponding vertex. Similar to the above *insert()* function, but takes as additional parameter the return values of a previous location query. See description of *Triangulation_3::locate()*.

The following method allows one to insert several points. It returns the number of inserted points.

template < class PointInputIterator >

std::ptrdiff_t *dt.insert(PointInputIterator first, PointInputIterator last)*

Inserts the points in the iterator range $[first, last)$. Returns the number of inserted points. Note that this function is not guaranteed to insert the points following the order of *PointInputIterator*, as *spatial_sort* is used to improve efficiency.
Precondition: The *value_type* of *first* and *last* is *Point*.

template < class PointWithInfoInputIterator >
std::ptrdiff_t *dt.insert(PointWithInfoInputIterator first, PointWithInfoInputIterator last)*

Inserts the points in the iterator range $[first, last)$. Returns the number of inserted points. Note that this function is not guaranteed to insert the points following the order of *PointWithInfoInputIterator*, as *spatial_sort* is used to improve efficiency. Given a pair *(p,i)*, the vertex *v* storing *p* also stores *i*, that is *v.point() == p* and *v.info() == i*. If several pairs have the same point, only one vertex is created, and one of the objects of type *Vertex::Info* will be stored in the vertex.
Precondition: *Vertex* must be model of the concept *TriangulationVertexBaseWithInfo_3*. The *value_type* of *first* and *last* is *std::pair<Point,Vertex::Info>*.

Displacement

Vertex_handle *dt.move_if_no_collision(Vertex_handle v, Point p)*

if there is not already another vertex placed on *p*, the triangulation is modified such that the new position of vertex *v* is *p*, and *v* is returned. Otherwise, the triangulation is not modified and the vertex at point *p* is returned.
Precondition: Vertex *v* must be finite.

Vertex_handle *dt.move(Vertex_handle v, Point p)*

same as above if there is no collision. Otherwise, *v* is deleted and the vertex placed on *p* is returned.
Precondition: Vertex *v* must be finite.

Removal

When a vertex *v* is removed from a triangulation, all the cells incident to *v* must be removed, and the polyhedral region consisting of all the tetrahedra that are incident to *v* must be re-triangulated. So, the problem reduces to triangulating a polyhedral region, while preserving its boundary, or to compute a *constrained* triangulation. This is known to be sometimes impossible: the Schönhardt polyhedron cannot be triangulated [She98a].

However, when dealing with Delaunay triangulations, the case of such polyhedra that cannot be re-triangulated cannot happen, so CGAL proposes a vertex removal.

void *dt.remove(Vertex_handle v)*

Removes the vertex *v* from the triangulation.
Precondition: *v* is a finite vertex of the triangulation.

template < typename InputIterator >
int *dt.remove(InputIterator first, InputIterator beyond)*

Removes the vertices specified by the iterator range [*first, beyond*) of value type *Vertex_handle*. *remove()* is called over each element of the range. The number of vertices removed is returned.
Precondition: (i) all vertices of the range are finite vertices of the triangulation; and (ii) no vertices are repeated in the range.

template < typename InputIterator >
int *dt.remove_cluster(InputIterator first, InputIterator beyond)*

This function has exactly the same result and the same preconditions as *remove(first, beyond)*. The difference is in the implementation and efficiency. This version does not re-triangulate the hole after each point removal but only after removing all vertices. This is more efficient if (and only if) the removed points are organized in a small number of connected components of the Delaunay triangulation.

If, due to some point removals, the size of the Delaunay triangulation decreases drastically, it might be interesting to defragment the *CGAL::Compact_container* (used by the *Triangulation_data_structure_3*).

Queries

Bounded_side *dt.side_of_sphere(Cell_handle c, Point p) const*

Returns a value indicating on which side of the circumscribed sphere of c the point p lies. More precisely, it returns:
- *ON_BOUNDED_SIDE* if p is inside the sphere. For an infinite cell this means that p lies strictly either in the half space limited by its finite facet and not containing any other point of the triangulation, or in the interior of the disk circumscribing the *finite* facet.
- *ON_BOUNDARY* if p on the boundary of the sphere. For an infinite cell this means that p lies on the circle circumscribing the *finite* facet.
- *ON_UNBOUNDED_SIDE* if p lies outside the sphere. For an infinite cell this means that p does not satisfy either of the two previous conditions.
Precondition: $dt.dimension() = 3$.

Bounded_side *dt.side_of_circle(Facet f, Point p) const*

Returns a value indicating on which side of the circumscribed circle of f the point p lies. More precisely, it returns:
- in dimension 3:
– For a finite facet, *ON_BOUNDARY* if p lies on the circle, *ON_UNBOUNDED_SIDE* when it lies in the exterior of the disk, *ON_BOUNDED_SIDE* when it lies in its interior.
– For an infinite facet, it considers the plane defined by the finite facet of the same cell, and does the same as in dimension 2 in this plane.
- in dimension 2:
– For a finite facet, *ON_BOUNDARY* if p lies on the circle, *ON_UNBOUNDED_SIDE* when it lies in the exterior of the disk, *ON_BOUNDED_SIDE* when it lies in its interior.
– For an infinite facet, *ON_BOUNDARY* if the point lies on the finite edge of f (endpoints included), *ON_BOUNDED_SIDE* for a point in the open half plane defined by f and not containing any other point of the triangulation, *ON_UNBOUNDED_SIDE* elsewhere.
Precondition: *dt.dimension()* ≥ 2 and in dimension 3, p is coplanar with f.

Bounded_side *dt.side_of_circle(Cell_handle c, int i, Point p)*

Same as the previous method for facet i of cell c.

Vertex_handle *dt.nearest_vertex(Point p, Cell_handle c = Cell_handle())*

Returns any nearest vertex to the point p, or the default constructed handle if the triangulation is empty. The optional argument c is a hint specifying where to start the search.
Precondition: c is a cell of *dt*.

Vertex_handle *dt.nearest_vertex_in_cell(Point p, Cell_handle c)*

Returns the vertex of the cell c that is nearest to p.

A point p is said to be in conflict with a cell c in dimension 3 (resp. a facet f in dimension 2) iff *dt.side_of_sphere(c, p)* (resp. *dt.side_of_circle(f, p)*) returns *ON_BOUNDED_SIDE*. The set of cells (resp. facets in dimension 2) which are in conflict with p is connected, and it forms a hole.

template <class OutputIteratorBoundaryFacets, class OutputIteratorCells>
std::pair<OutputIteratorBoundaryFacets, OutputIteratorCells>

dt.find_conflicts(Point p,
Cell_handle c,
OutputIteratorBoundaryFacets bfit,

OutputIteratorCells cit)

Computes the conflict hole induced by *p*. The starting cell (resp. facet) *c* must be in conflict. Then this function returns respectively in the output iterators:
– *cit*: the cells (resp. facets) in conflict.
– *bfit*: the facets (resp. edges) on the boundary, that is, the facets (resp. edges) *(t, i)* where the cell (resp. facet) *t* is in conflict, but *t->neighbor(i)* is not.
This function can be used in conjunction with *insert_in_hole()* in order to decide the insertion of a point after seeing which elements of the triangulation are affected. Returns the pair composed of the resulting output iterators.
Precondition: *dt.dimension()* ≥ 2, and *c* is in conflict with *p*.

template <class OutputIteratorBoundaryFacets, class OutputIteratorCells, class OutputIteratorInternalFacets>

Triple<OutputIteratorBoundaryFacets, OutputIteratorCells, OutputIteratorInternalFacets>

dt.find_conflicts(Point p,
Cell_handle c,
OutputIteratorBoundaryFacets bfit,
OutputIteratorCells cit,
OutputIteratorInternalFacets ifit)

Same as the other *find_conflicts()* function, except that it also computes the internal facets, i.e. the facets common to two cells which are in conflict with *p*. Then this function returns respectively in the output iterators:
– *cit*: the cells (resp. facets) in conflict.
– *bfit*: the facets (resp. edges) on the boundary, that is, the facets (resp. edges) *(t, i)* where the cell (resp. facet) *t* is in conflict, but *t->neighbor(i)* is not.
– *ifit*: the facets (resp. edges) inside the hole, that is, delimiting two cells (resp facets) in conflict.
Returns the *Triple* composed of the resulting output iterators.
Precondition: *dt.dimension()* ≥ 2, and *c* is in conflict with *p*.

deprecated

template <class OutputIterator>
OutputIterator *dt.vertices_in_conflict(Point p, Cell_handle c, OutputIterator res)*

This function is renamed *vertices_on_conflict_zone_boundary* since CGAL-3.8.

deprecated

template <class OutputIterator>
OutputIterator *dt.vertices_on_conflict_zone_boundary(Point p, Cell_handle c, OutputIterator res)*

Similar to *find_conflicts()*, but reports the vertices which are on the boundary of the conflict hole of *p*, in the output iterator *res*. Returns the resulting output iterator.
Precondition: *dt.dimension()* ≥ 2, and *c* is in conflict with *p*.

A face (cell, facet or edge) is said to be a Gabriel face iff its smallest circumscribing sphere do not enclose any vertex of the triangulation. Any Gabriel face belongs to the Delaunay triangulation, but the reciprocal is not true. The following member functions test the Gabriel property of Delaunay faces.
bool *dt.is_Gabriel(Cell_handle c, int i)*

bool	*dt.is_Gabriel(Cell_handle c, int i, int j)*
bool	*dt.is_Gabriel(Facet f)*
bool	*dt.is_Gabriel(Edge e)*

Voronoi diagram

CGAL offers several functionalities to display the Voronoi diagram of a set of points in 3D.

Note that the user should use a kernel with exact constructions in order to guarantee the computation of the Voronoi diagram (as opposed to computing the triangulation only, which requires only exact predicates).

Point *dt.dual(Cell_handle c) const*

Returns the circumcenter of the four vertices of c.
Precondition: *dt.dimension()*$=3$ and c is not infinite.

Object *dt.dual(Facet f) const*

Returns the dual of facet f, which is
in dimension 3: either a segment, if the two cells incident to f are finite, or a ray, if one of them is infinite;
in dimension 2: a point.
Precondition: *dt.dimension()* ≥ 2 and f is not infinite.

Object *dt.dual(Cell_handle c, int i) const*

same as the previous method for facet *(c,i)*.

Line *dt.dual_support(Cell_handle c, int i) const*

returns the line supporting the dual of the facet.
Precondition: *dt.dimension()* ≥ 2 and f is not infinite.

template <class Stream>
Stream& *dt.draw_dual(Stream & os)*

Sends the set of duals to all the facets of *dt* into *os*.

debugging support

Checking

bool *dt.is_valid(bool verbose = false) const*

Checks the combinatorial validity of the triangulation and the validity of its geometric embedding (see Section 39.1). Also checks that all the circumscribing spheres (resp. circles in dimension 2) of cells (resp. facets in dimension 2) are empty.
When *verbose* is set to true, messages describing the first invalidity encountered are printed.

bool *dt.is_valid(Cell_handle c, bool verbose = false) const*

Checks the combinatorial and geometric validity of the cell (see Section 39.1). Also checks that the circumscribing sphere (resp. circle in dimension 2) of cells (resp. facet in dimension 2) is empty.
When *verbose* is set to true, messages are printed to give a precise indication of the kind of invalidity encountered.

These methods are mainly a debugging help for the users of advanced features.

debugging support

See Also

CGAL::Regular_triangulation_3

CGAL::Triangulation_hierarchy_3<Tr>

deprecated

Note : this class is deprecated since CGAL 3.6. Its functionality has been replaced by the use of the *Fast_location* tag as the *LocationPolicy* template parameter in *Delaunay_triangulation_3*.

Definition

The class *Triangulation_hierarchy_3* implements a triangulation augmented with a data structure which allows fast point location queries. As proved in [Dev02], this structure has an optimal behavior when it is built for Delaunay triangulations. It can however be used for other triangulations.

#include <CGAL/Triangulation_hierarchy_3.h>

Parameters

It is templated by a parameter which must be instantiated by one of the CGAL triangulation classes. *In the current implementation, only Delaunay_triangulation_3 is supported for Tr.*

Tr::Vertex has to be a model of the concept *TriangulationHierarchyVertexBase_3*.
Tr::Geom_traits has to be a model of the concept *DelaunayTriangulationTraits_3*.

Inherits From

Tr

Triangulation_hierarchy_3<Tr> offers exactly the same functionalities as *Tr*. Most of them (point location, insertion, removal...) are overloaded to improve their efficiency by using the hierarchic structure.

Note that, since the algorithms that are provided are randomized, the running time of constructing a triangulation with a hierarchy may be improved when shuffling the data points.

However, the I/O operations are not overloaded. So, writing a hierarchy into a file will lose the hierarchic structure and reading it from the file will result in an ordinary triangulation whose efficiency will be the same as *Tr*.

Implementation

The data structure is a hierarchy of triangulations. The triangulation at the lowest level is the original triangulation where operations and point location are to be performed. Then at each succeeding level, the data structure stores a triangulation of a small random sample of the vertices of the triangulation at the preceding level. Point location is done through a top-down nearest neighbor query. The nearest neighbor query is first performed naively in the top level triangulation. Then, at each following level, the nearest neighbor at that level is found through a linear walk performed from the nearest neighbor found at the preceding level. Because the number of vertices in each triangulation is only a small fraction of the number of vertices of the preceding triangulation the data structure remains small and achieves fast point location queries on real data.

See Also

CGAL::Triangulation_hierarchy_vertex_base_3
CGAL::Delaunay_triangulation_3

deprecated

CGAL::Regular_triangulation_3<RegularTriangulationTraits_3,TriangulationDataStructure_3>

Definition

Let $S^{(w)}$ be a set of weighted points in $\mathbb{R}^3$. Let $p^{(w)} = (p, w_p), p \in \mathbb{R}^3, w_p \in \mathbb{R}$ and $z^{(w)} = (z, w_z), z \in \mathbb{R}^3, w_z \in \mathbb{R}$ be two weighted points. A weighted point $p^{(w)} = (p, w_p)$ can also be seen as a sphere of center p and radius $\sqrt{w_p}$. The *power product* (or *power distance*) between $p^{(w)}$ and $z^{(w)}$ is defined as

$$\Pi(p^{(w)}, z^{(w)}) = \|p - z\|^2 - w_p - w_z$$

where $\|p - z\|$ is the Euclidean distance between p and z. $p^{(w)}$ and $z^{(w)}$ are said to be *orthogonal* if $\Pi(p^{(w)} - z^{(w)}) = 0$ (see Figure 39.2).

Four weighted points have a unique common orthogonal weighted point called the *power sphere*. A sphere $z^{(w)}$ is said to be *regular* if $\forall p^{(w)} \in S^{(w)}, \Pi(p^{(w)} - z^{(w)}) \geq 0$.

A triangulation of $S^{(w)}$ is *regular* if the power spheres of all simplices are regular.

#include <CGAL/Regular_triangulation_3.h>

Parameters

The first template argument must be a model of the *RegularTriangulationTraits_3* concept.

The second template argument must be a model of the *TriangulationDataStructure_3* concept. It has the default value *Triangulation_data_structure_3<Triangulation_vertex_base_3<RegularTriangulationTraits_3>, Regular_triangulation_cell_base_3<RegularTriangulationTraits_3> >*.

Inherits From

Triangulation_3<RegularTriangulationTraits_3,TriangulationDataStructure_3>

Types

typedef RegularTriangulationTraits_3::Bare_point	*Bare_point;*	The type for points p of weighted points $p^{(w)} = (p, w_p)$
typedef RegularTriangulationTraits_3::Weighted_point_3	*Weighted_point;*	

Creation

Regular_triangulation_3<RegularTriangulationTraits_3,TriangulationDataStructure_3> rt(RegularTriangulationTraits_3 traits = RegularTriangulationTraits_3())

Creates an empty regular triangulation, possibly specifying a traits class *traits*.

Regular_triangulation_3<RegularTriangulationTraits_3,TriangulationDataStructure_3> rt(Regular_triangulation_3 rt1)

Copy constructor.

template < class InputIterator >
Regular_triangulation_3<RegularTriangulationTraits_3,TriangulationDataStructure_3> rt(InputIterator first, InputIterator last, RegularTriangulationTraits_3 traits = RegularTriangulationTraits_3())

Equivalent to constructing an empty triangulation with the optional traits class argument and calling *insert(first,last)*.

Operations

Insertion

The following methods, which already exist in *Triangulation_3*, are overloaded to ensure the property that all power spheres are regular.

Vertex_handle *rt.insert(Weighted_point p, Cell_handle start = Cell_handle())*

Inserts weighted point p in the triangulation. The optional argument *start* is used as a starting place for the search.
If this insertion creates a vertex, this vertex is returned.
If p coincides with an existing vertex and has a greater weight, then the existing weighted point becomes hidden (see *RegularTriangulationCellBase_3*) and p replaces it as vertex of the triangulation.
If p coincides with an already existing vertex (both point and weights being equal), then this vertex is returned and the triangulation remains unchanged.
Otherwise if p does not appear as a vertex of the triangulation, then it is stored as a hidden point and this method returns the default constructed handle.

Vertex_handle *rt.insert(Weighted_point p, Vertex_handle hint)*

Same as above but uses *hint* as a starting place for the search.

Vertex_handle *rt.insert(Weighted_point p, Locate_type lt, Cell_handle loc, int li, int lj)*

Inserts weighted point p in the triangulation and returns the corresponding vertex. Similar to the above *insert()* function, but takes as additional parameter the return values of a previous location query. See description of *Triangulation_3::locate()*.

The following method allows one to insert several points.

template < class InputIterator >

std::ptrdiff_t *rt.insert(InputIterator first, InputIterator last)*

Inserts the weighted points in the range [*first*, *last*). It returns the difference of the number of vertices between after and before the insertions (it may be negative due to hidden points). Note that this function is not guaranteed to insert the points following the order of *InputIterator*, as *spatial_sort* is used to improve efficiency.
Precondition: The *value_type* of *first* and *last* is *Weighted_point*.

template < class WeightedPointWithInfoInputIterator >
std::ptrdiff_t *rt.insert(WeightedPointWithInfoInputIterator first, WeightedPointWithInfoInputIterator last)*

Inserts the weighted points in the iterator range [*first*, *last*). It returns the difference of the number of vertices between after and before the insertions (it may be negative due to hidden points). Note that this function is not guaranteed to insert the weighted points following the order of *WeightedPointWithInfoInputIterator*, as *spatial_sort* is used to improve efficiency. Given a pair *(p,i)*, the vertex *v* storing *p* also stores *i*, that is *v.point() == p* and *v.info() == i*. If several pairs have the same point, only one vertex is created, one of the objects of type *Vertex::Info* will be stored in the vertex.
Precondition: *Vertex* must be model of the concept *TriangulationVertexBaseWithInfo_3*. The *value_type* of *first* and *last* is *std::pair<Weighted_point,Vertex::Info>*.

The following methods, which already exist in *Triangulation_3*, are overloaded to ensure that hidden points are well created and maintained.

template <class CellIt>
Vertex_handle *rt.insert_in_hole(Weighted_point p, CellIt cell_begin, CellIt cell_end, Cell_handle begin, int i)*

Creates a new vertex by starring a hole. It takes an iterator range [*cell_begin*; *cell_end*[of *Cell_handle*s which specifies a hole: a set of connected cells (resp. facets in dimension 2) which is star-shaped wrt *p*. (*begin*, *i*) is a facet (resp. an edge) on the boundary of the hole, that is, *begin* belongs to the set of cells (resp. facets) previously described, and *begin->neighbor(i)* does not. Then this function deletes all the cells (resp. facets) describing the hole, creates a new vertex *v*, and for each facet (resp. edge) on the boundary of the hole, creates a new cell (resp. facet) with *v* as vertex. Then *v->set_point(p)* is called and *v* is returned.
If the hole contains interior vertices, each of them is hidden by the insertion of *p* and is stored in the new cell which contains it.
Precondition: *rt.dimension()* ≥ 2, the set of cells (resp. facets in dimension 2) is connected, not empty, its boundary is connected, and *p* lies inside the hole, which is star-shaped wrt *p*.

template <class CellIt>
Vertex_handle *rt.insert_in_hole(Weighted_point p,*

CellIt cell_begin,
CellIt cell_end,
Cell_handle begin,
int i,
Vertex_handle newv)

Same as above, except that *newv* will be used as the new vertex, which must have been allocated previously with e.g. *create_vertex*.

Removal

void *rt.remove(Vertex_handle v)*

Removes the vertex *v* from the triangulation.

Queries

Let us remark that

$$\Pi(p^{(w)} - z^{(w)}) > 0$$

is equivalent to

p lies outside the sphere with center z and radius $\sqrt{w_p^2 + w_z^2}$.

This remark helps provide an intuition about the following predicates.

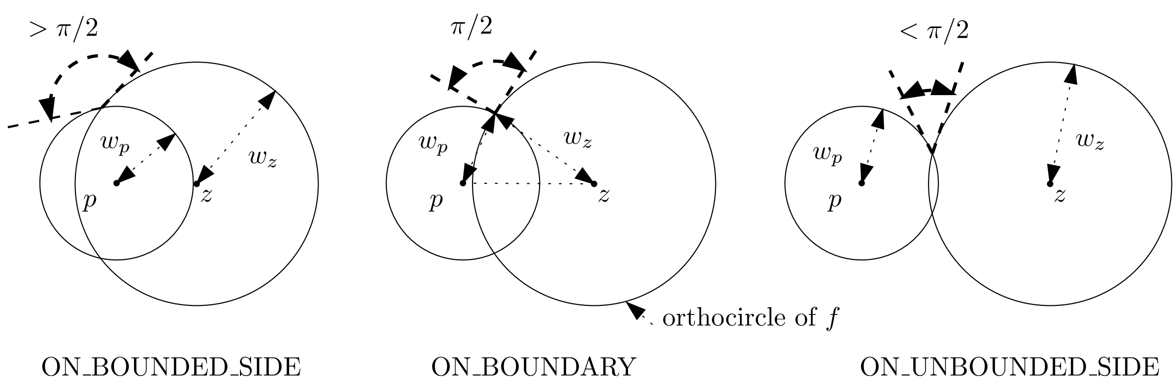


Figure 39.13: side_of_power_circle.

Bounded_side *rt.side_of_power_sphere(Cell_handle c, Weighted_point p) const*

Returns the position of the weighted point p with respect to the power sphere of c. More precisely, it returns:
- *ON_BOUNDED_SIDE* if $\Pi(p^{(w)} - z(c)^{(w)}) < 0$ where $z(c)^{(w)}$ is the power sphere of c. For an infinite cell this means either that p lies strictly in the half space limited by its finite facet and not containing any other point of the triangulation, or that the angle between p and the power circle of the *finite* facet of c is greater than $\pi/2$.
- *ON_BOUNDARY* if p is orthogonal to the power sphere of c i.e. $\Pi(p^{(w)} - z(c)^{(w)}) = 0$. For an infinite cell this means that p is orthogonal to the power circle of its *finite* facet.
- *ON_UNBOUNDED_SIDE* if $\Pi(p^{(w)} - z(c)^{(w)}) > 0$ i.e. the angle between the weighted point p and the power sphere of c is less than $\pi/2$ or if these two spheres do not intersect. For an infinite cell this means that p does not satisfy either of the two previous conditions.
Precondition: *rt.dimension()* $= 3$.

Bounded_side *rt.side_of_power_circle(Facet f, Weighted_point p) const*

Returns the position of the point p with respect to the power circle of f. More precisely, it returns:
— in dimension 3:
– For a finite facet,
ON_BOUNDARY if p is orthogonal to the power circle in the plane of the facet,
ON_UNBOUNDED_SIDE when their angle is less than $\pi/2$,
ON_BOUNDED_SIDE when it is greater than $\pi/2$ (see Figure 39.13).
– For an infinite facet, it considers the plane defined by the finite facet of the cell *f.first*, and does the same as in dimension 2 in this plane.
— in dimension 2:
– For a finite facet,
ON_BOUNDARY if p is orthogonal to the circle,
ON_UNBOUNDED_SIDE when the angle between p and the power circle of f is less than $\pi/2$, *ON_BOUNDED_SIDE* when it is greater than $\pi/2$.
– For an infinite facet,
ON_BOUNDED_SIDE for a point in the open half plane defined by f and not containing any other point of the triangulation,
ON_UNBOUNDED_SIDE in the other open half plane.
If the point p is collinear with the finite edge e of f, it returns:
ON_BOUNDED_SIDE if $\Pi(p^{(w)} - z(e)^{(w)}) < 0$, where $z(e)^{(w)}$ is the power segment of e in the line supporting e,
ON_BOUNDARY if $\Pi(p^{(w)} - z(e)^{(w)}) = 0$,
ON_UNBOUNDED_SIDE if $\Pi(p^{(w)} - z(e)^{(w)}) > 0$.
Precondition: *rt.dimension()* ≥ 2.

Bounded_side *rt.side_of_power_circle(Cell_handle c, int i, Weighted_point p) const*

Same as the previous method for facet i of cell c.

Bounded_side *rt.side_of_power_segment(Cell_handle c, Weighted_point p) const*

In dimension 1, returns
ON_BOUNDED_SIDE if $\Pi(p^{(w)} - z(c)^{(w)}) < 0$, where $z(c)^{(w)}$ is the power segment of the edge represented by c,
ON_BOUNDARY if $\Pi(p^{(w)} - z(c)^{(w)}) = 0$,
ON_UNBOUNDED_SIDE if $\Pi(p^{(w)} - z(c)^{(w)}) > 0$.
Precondition: *rt.dimension()* $= 1$.

Vertex_handle *rt.nearest_power_vertex(Weighted_point p, Cell_handle c = Cell_handle())*

Returns the vertex of the triangulation which is nearest to p with respect to the power distance. This means that the power of the query point p with respect to the weighted point in the returned vertex is smaller than the power of p with respect to the weighted point in any other vertex. Ties are broken arbitrarily. The default constructed handle is returned if the triangulation is empty. The optional argument c is a hint specifying where to start the search.
Precondition: c is a cell of *rt*.

Vertex_handle *rt.nearest_power_vertex_in_cell(Weighted_point p, Cell_handle c)*

Returns the vertex of the cell c that is nearest to p with respect to the power distance.

A weighted point p is said to be in conflict with a cell c in dimension 3 (resp. with a facet f in dimension 2) if it has a negative power distance to the power sphere of c (resp. to the power circle of f). The set of cells (resp. facets in dimension 2) which are in conflict with p is connected.

template <class OutputIteratorBoundaryFacets, class OutputIteratorCells, class OutputIteratorInternalFacets>

Triple<OutputIteratorBoundaryFacets, OutputIteratorCells, OutputIteratorInternalFacets>

rt.find_conflicts(const Weighted_point p,
Cell_handle c,
OutputIteratorBoundaryFacets bfit,
OutputIteratorCells cit,
OutputIteratorInternalFacets ifit)

Compute the conflicts with p. The starting cell (resp. facet) c must be in conflict with p. Then this function returns respectively in the output iterators:
– *cit*: the cells (resp. facets) in conflict with p.
– *bfit*: the facets (resp. edges) on the boundary of the conflict zone, that is, the facets (resp. edges) *(t, i)* where the cell (resp. facet) t is in conflict, but *t->neighbor(i)* is not.
– *ifit*: the facets (resp. edges) inside the conflict zone, that facets incident to two cells (resp facets) in conflict.
Returns the *Triple* composed of the resulting output iterators.
Precondition: *rt.dimension()* ≥ 2, and c is in conflict with p.

deprecated

template <class OutputIterator>

OutputIterator *rt.vertices_in_conflict(Weighted_point p, Cell_handle c, OutputIterator res)*

This function is renamed *vertices_on_conflict_zone_boundary* since CGAL-3.8.

deprecated

template <class OutputIterator>
OutputIterator *rt.vertices_on_conflict_zone_boundary(Weighted_point p, Cell_handle c, OutputIterator res)*

Similar to *find_conflicts()*, but reports the vertices which are on the boundary of the conflict zone of *p*, in the output iterator *res*. Returns the resulting output iterator.
Precondition: *rt.dimension()* ≥ 2, and *c* is a cell containing *p*.

template <class OutputIterator>
OutputIterator *rt.vertices_inside_conflict_zone(Weighted_point p, Cell_handle c, OutputIterator res)*

Similar to *find_conflicts()*, but reports the vertices which are in the interior of the conflict zone of *p*, in the output iterator *res*. The vertices that are on the boundary of the conflict zone are not reported. Returns the resulting output iterator.
Precondition: *rt.dimension()* ≥ 2, and *c* is a cell containing *p*.

In the weighted setting, a face (cell, facet, edge or vertex) is said to be a Gabriel face iff the smallest sphere orthogonal to the weighted points associated to its vertices, has a positive power product with the weighted point of any other vertex of the triangulation. Any weighted Gabriel face belongs to the regular triangulation, but the reciprocal is not true. The following member functions test the Gabriel property of the faces of the regular triangulation.

bool *rt.is_Gabriel(Cell_handle c, int i)*

bool *rt.is_Gabriel(Cell_handle c, int i, int j)*
bool *rt.is_Gabriel(Facet f)*
bool *rt.is_Gabriel(Edge e)*
bool *rt.is_Gabriel(Vertex_handle v)*

Power diagram

CGAL offers several functionalities to display the Power diagram of a set of points in 3D.

Note that the user should use a kernel with exact constructions in order to guarantee the computation of the Voronoi diagram (as opposed to computing the triangulation only, which requires only exact predicates).

Weighted_point

rt.dual(Cell_handle c) const

Returns the weighted circumcenter of the four vertices of c.
Precondition: *rt.dimension()*$= 3$ and *c* is not infinite.

Object *rt.dual(Facet f) const*

Returns the dual of facet f, which is
in dimension 3: either a segment, if the two cells incident to f are finite, or a ray, if one of them is infinite;
in dimension 2: a point.
Precondition: *rt.dimension()* ≥ 2 and f is not infinite.

Object *rt.dual(Cell_handle c, int i) const*

same as the previous method for facet *(c,i)*.

template <class Stream>
Stream& *rt.draw_dual(Stream & os)*

Sends the set of duals to all the facets of *rt* into *os*.

debugging support

Checking

bool *rt.is_valid(bool verbose = false) const*

Checks the combinatorial validity of the triangulation and the validity of its geometric embedding (see Section 39.1). Also checks that all the power spheres (resp. power circles in dimension 2, power segments in dimension 1) of cells (resp. facets in dimension 2, edges in dimension 1) are regular. When *verbose* is set to true, messages describing the first invalidity encountered are printed.
This method is mainly a debugging help for the users of advanced features.

debugging support

Concept

TriangulationTraits_3

Definition

The concept TriangulationTraits_3 is the first template parameter of the class *Triangulation_3*. It defines the geometric objects (points, segments, triangles and tetrahedra) forming the triangulation together with a few geometric predicates and constructions on these objects : lexicographical comparison, orientation in case of coplanar points and orientation in space.

Types

TriangulationTraits_3:: Point_3	The point type. It must be *DefaultConstructible*, *CopyConstructible* and *Assignable*.
TriangulationTraits_3:: Segment_3	The segment type.
TriangulationTraits_3:: Tetrahedron_3	The tetrahedron type.
TriangulationTraits_3:: Triangle_3	The triangle type.

TriangulationTraits_3:: Construct_segment_3

A constructor object that must provide the function operator
Segment_3 operator()(Point_3 p, Point_3 q),
which constructs a segment from two points.

TriangulationTraits_3:: Construct_triangle_3

A constructor object that must provide the function operator
Triangle_3 operator()(Point_3 p, Point_3 q, Point_3 r),
which constructs a triangle from three points.

TriangulationTraits_3:: Construct_tetrahedron_3

A constructor object that must provide the function operator
Tetrahedron_3 operator()(Point_3 p, Point_3 q, Point_3 r, Point_3 s),
which constructs a tetrahedron from four points.

TriangulationTraits_3:: Compare_xyz_3

A predicate object that must provide the function operator
Comparison_result operator()(Point p, Point q),
which returns *EQUAL* if the two points are equal. Otherwise it must return a consistent order for any two points chosen in a same line.

TriangulationTraits_3:: Coplanar_orientation_3

A predicate object that must provide the function operator
Orientation operator()(Point p, Point q, Point r),
which returns *COLLINEAR* if the points are collinear. Otherwise it must return a consistent orientation for any three points chosen in a same plane.

TriangulationTraits_3:: Orientation_3

A predicate object that must provide the function operator
Orientation operator()(Point p, Point q, Point r, Point s),
which returns POSITIVE, if s lies on the positive side of the oriented plane h defined by p, q, and r, returns NEGATIVE if s lies on the negative side of h, and returns COPLANAR if s lies on h.

Creation

TriangulationTraits_3 traits;	Default constructor.
TriangulationTraits_3 traits(Triangulation_traits_3 tr);	Copy constructor.

Operations

The following functions give access to the predicate and construction objects:

Construct_tetrahedron_3	*traits.construct_tetrahedron_3_object()*
Construct_triangle_3	*traits.construct_triangle_3_object()*
Construct_segment_3	*traits.construct_segment_3_object()*
Compare_xyz_3	*traits.compare_xyz_3_object()*
Coplanar_orientation_3	*traits.coplanar_orientation_3_object()*
Orientation_3	*traits.orientation_3_object()*

Has Models

CGAL::Exact_predicates_inexact_constructions_kernel (recommended)
CGAL::Exact_predicates_exact_constructions_kernel
CGAL::Filtered_kernel
CGAL::Cartesian
CGAL::Simple_cartesian
CGAL::Homogeneous
CGAL::Simple_homogeneous

Concept

DelaunayTriangulationTraits_3

Definition

The concept DelaunayTriangulationTraits_3 is the first template parameter of the class *Delaunay_triangulation_3*. It defines the geometric objects (points, segments...) forming the triangulation together with a few geometric predicates and constructions on these objects.

Refines

TriangulationTraits_3

In addition to the requirements described for the traits class of *Triangulation_3*, the geometric traits class of a Delaunay triangulation must fulfill the following requirements:

Types

DelaunayTriangulationTraits_3:: Line_3	The line type.
DelaunayTriangulationTraits_3:: Object_3	The object type.
DelaunayTriangulationTraits_3:: Ray_3	The ray type.

DelaunayTriangulationTraits_3:: Coplanar_side_of_bounded_circle_3

A predicate object that must provide the function operator
Bounded_side operator()(Point p, Point q, Point r, Point s),
which determines the bounded side of the circle defined by p, q, and r on which s lies.
Precondition: p, q, r, and s are coplanar and p, q, and r are not collinear.

DelaunayTriangulationTraits_3:: Side_of_oriented_sphere_3

A predicate object that must provide the function operator
Oriented_side operator()(Point p, Point q, Point r, Point s, Point t),
which determines on which side of the oriented sphere circumscribing p, q, r, s the point t lies.

DelaunayTriangulationTraits_3:: Compare_distance_3

A predicate object that must provide the function operator
Comparison_result operator()(Point p, Point q, Point r),
which compares the distance between p and q to the distance between p and r. It is only needed when using the *Fast_location* policy or the *nearest_vertex* function.

In addition, only when the dual operations are used, the traits class must provide the following constructor objects:

DelaunayTriangulationTraits_3:: Construct_circumcenter_3

A constructor object that must provide the function operator
Point_3 operator()(Point_3 p, Point_3 q, Point_3 r, Point_3 s),
which constructs the circumcenter of four points.
Precondition: p, q, r and s must be non coplanar.
It must also provide the function operator
Point_3 operator()(Point_3 p, Point_3 q, Point_3 r),
which constructs the circumcenter of three points.
Precondition: p, q and r must be non collinear.

DelaunayTriangulationTraits_3:: Construct_object_3

A constructor object that must provide the function operators
Object_3 operator()(Point_3 p),
Object_3 operator()(Segment_3 s) and
Object_3 operator()(Ray_3 r)
that construct an object respectively from a point, a segment and a ray.

DelaunayTriangulationTraits_3:: Construct_equidistant_line_3

A constructor object that must provide the function operator
Line_3 operator()(Point_3 p1, Point_3 p2, Point_3 p3),
which constructs the line which is at the same distance from the three points.
Precondition: *p1*, *p2* and *p3* must be non collinear.

DelaunayTriangulationTraits_3:: Construct_ray_3

A constructor object that must provide the function operator
Ray_3 operator()(Point_3 p, Line_3 l),
which constructs the ray starting at p with direction given by l.

Operations

The following functions give access to the predicate and construction objects:

Coplanar_side_of_bounded_circle_3	*traits.coplanar_side_of_bounded_circle_3_object()*
Side_of_oriented_sphere_3	*traits.side_of_oriented_sphere_3_object()*

When using the *Fast_location* policy or the *nearest_vertex* function, the traits must provide:

Compare_distance_3	*traits.compare_distance_3_object()*

The following functions must be provided only if the methods of *Delaunay_triangulation_3* returning elements of the Voronoi diagram are instantiated:

Construct_circumcenter_3	*traits.construct_circumcenter_3_object()*
Construct_object_3	*traits.construct_object_3_object()*
Construct_perpendicular_line_3	*traits.construct_perpendicular_line_object()*
Construct_plane_3	*traits.construct_plane_3_object()*
Construct_ray_3	*traits.construct_ray_3_object()*

Has Models

CGAL::Exact_predicates_inexact_constructions_kernel (recommended)
CGAL::Exact_predicates_exact_constructions_kernel (recommended for Voronoi)
CGAL::Filtered_kernel
CGAL::Cartesian
CGAL::Simple_cartesian
CGAL::Homogeneous
CGAL::Simple_homogeneous

RegularTriangulationTraits_3

Concept

Definition

The concept RegularTriangulationTraits_3 is the first template parameter of the class *Regular_triangulation_3*. It defines the geometric objects (points, segments...) forming the triangulation together with a few geometric predicates and constructions on these objects.

Refines

TriangulationTraits_3

In addition to the requirements described for the traits class of *Triangulation_3*, the geometric traits class of *Regular_triangulation_3* must fulfill the following requirements.

Types

RegularTriangulationTraits_3:: Line_3	The line type.
RegularTriangulationTraits_3:: Object_3	The object type.
RegularTriangulationTraits_3:: Plane_3	The plane type.
RegularTriangulationTraits_3:: Ray_3	The ray type.

We use here the same notation as in Section 39.3. To simplify notation, p will often denote in the sequel either the point $p \in \mathbb{R}^3$ or the weighted point $p^{(w)} = (p, w_p)$.

RegularTriangulationTraits_3:: Weighted_point_3	The weighted point type.
RegularTriangulationTraits_3:: Bare_point	The (un-weighted) point type.

RegularTriangulationTraits_3:: Power_test_3

A predicate object which must provide the following function operators:

Oriented_side operator()(Weighted_point_3 p, Weighted_point_3 q, Weighted_point_3 r, Weighted_point_3 s, Weighted_point_3 t),
which performs the following:
Let $z(p,q,r,s)^{(w)}$ be the power sphere of the weighted points (p,q,r,s). Returns
ON_ORIENTED_BOUNDARY if t is orthogonal to $z(p,q,r,s)^{(w)}$,
ON_NEGATIVE_SIDE if t lies outside the oriented sphere of center $z(p,q,r,s)$ and radius $\sqrt{w^2_{z(p,q,r,s)} + w^2_t}$ (which is equivalent to $\Pi(t^{(w)}, z(p,q,r,s)^{(w)} > 0)$),
ON_POSITIVE_SIDE if t lies inside this oriented sphere.
Precondition: *p, q, r, s* are not coplanar. Note that with this definition, if all the points have a weight equal to 0, then *power_test_3(p,q,r,s,t) = side_of_oriented_sphere(p,q,r,s,t)*.

Oriented_side operator()(Weighted_point_3 p, Weighted_point_3 q, Weighted_point_3 r, Weighted_point_3 t),
which has an definition analogous to the previous method, for coplanar points, with the power circle $z(p,q,r)^{(w)}$.
Precondition: *p, q, r* are not collinear and *p, q, r, t* are coplanar. If all the points have a weight equal to 0, then *power_test_3(p,q,r,t) = side_of_oriented_circle(p,q,r,t)*.

Oriented_side operator()(Weighted_point_3 p, Weighted_point_3 q, Weighted_point_3 t),
which is the same for collinear points, where $z(p,q)^{(w)}$ is the power segment of p and q.
Precondition: p and q have different Bare_points, and *p, q, t* are collinear. If all points have a weight equal to 0, then *power_test_3(p,q,t)* gives the same answer as the kernel predicate *s(p,q).has_on(t)* would give, where *s(p,q)* denotes the segment with endpoints p and q.

Oriented_side operator()(Weighted_point_3 p, Weighted_point_3 q),
which is the same for equal points, that is when p and q have equal coordinates, then it returns the comparison of the weights (*ON_POSITIVE_SIDE* when q is heavier than p).
Precondition: p and q have equal Bare_points.

The following predicate is required if a call to *nearest_power_vertex* or *nearest_power_vertex_in_cell* is issued:

RegularTriangulationTraits_3:: Compare_power_distance_3

A predicate object that must provide the function operator
Comparison_result operator()(Point_3 p, Weighted_point_3 q, Weighted_point_3 r),
which compares the power distance between p and q to the power distance between p and r.

In addition, only when the dual operations are used, the traits class must provide the following constructor objects:

RegularTriangulationTraits_3:: Construct_weighted_circumcenter_3

A constructor type. The operator() constructs the bare point which is the center of the smallest orthogonal sphere to the input weighted points.
Bare_point operator() (Weighted_point_3 p, Weighted_point_3 q, Weighted_point_3 r, Weighted_point_3 s);

RegularTriangulationTraits_3:: Construct_object_3

A constructor object that must provide the function operators
Object_3 operator()(Point_3 p),
Object_3 operator()(Segment_3 s) and
Object_3 operator()(Ray_3 r)
that construct an object respectively from a point, a segment and a ray.

RegularTriangulationTraits_3:: Construct_perpendicular_line_3

A constructor object that must provide the function operator
Line_3 operator()(Plane_3 pl, Point_3 p),
which constructs the line perpendicular to *pl* passing through *p*.

RegularTriangulationTraits_3:: Construct_plane_3

A constructor object that must provide the function operator
Plane_3 operator()(Point_3 p, Point_3 q, Point_3 r),
which constructs the plane passing through *p*, *q* and *r*.
Precondition: *p*, *q* and *r* are non collinear.

RegularTriangulationTraits_3:: Construct_ray_3

A constructor object that must provide the function operator
Ray_3 operator()(Point_3 p, Line_3 l),
which constructs the ray starting at *p* with direction given by *l*.

Operations

The following function gives access to the predicate object:

Power_test_3 *traits.power_test_3_object()*

The following functions must be provided only if the member functions of *Regular_triangulation_3* returning elements of the dual diagram are called:

Construct_weighted_circumcenter_3

traits.construct_weighted_circumcenter_3_object()
Construct_object_3 *traits.construct_object_3_object()*
Construct_perpendicular_line_3

traits.construct_perpendicular_line_object()
Construct_plane_3 *traits.construct_plane_3_object()*
Construct_ray_3 *traits.construct_ray_3_object()*

Has Models

CGAL::Regular_triangulation_euclidean_traits_3

CGAL::Regular_triangulation_euclidean_traits_3<K,Weight>

Definition

The class *Regular_triangulation_euclidean_traits_3<K,Weight>* is designed as a default traits class for the class *Regular_triangulation_3<RegularTriangulationTraits_3,TriangulationDataStructure_3>*. It provides *Weighted_point_3*, a class for weighted points, which derives from the three dimensional point class *K::Point_3*.

The first argument *K* must be a model of the *Kernel* concept.

The second argument *Weight* of the class *Regular_triangulation_euclidean_traits_3<K,Weight>* is in fact optional: if is it not provided, *K::RT* will be used.

The class is a model of the concept *RegularTriangulationTraits_3* but it also contains predicates and constructors on weighted points that are not required in the concept *RegularTriangulationTraits_3*.

Note that filtered predicates are automatically used if the boolean *Has_filtered_predicates* in the kernel provided as template parameter of that class is set to *true*.

#include <CGAL/Regular_triangulation_euclidean_traits_3.h>

Is Model for the Concepts

RegularTriangulationTraits_3

Inherits From

K

Types

typedef K::Point_3	*Bare_point;*	The type for point p of a weighted point $p^{(w)} = (p, w_p)$.
typedef Weighted_point <Bare_point, Weight>	*Weighted_point_3;*	The type for weighted points.

Types for predicate functors

Regular_triangulation_euclidean_traits_3<K,Weight>:: Power_test_3

A predicate type for power test. Belongs to the *RegularTriangulationTraits_3* concept.

Regular_triangulation_euclidean_traits_3<K,Weight>:: Compare_power_distance_3

A predicate type to compare power distance. Belongs to the *RegularTriangulationTraits_3* concept.

Regular_triangulation_euclidean_traits_3<K,Weight>:: Compare_weighted_squared_radius_3

A predicate type. The operator() takes weighted point(s) as arguments, together with one weight. It compares the weight of the smallest sphere orthogonal to the weighted points with the input weight.
Comparison_result operator()(Weighted_point_3 p, Weighted_point_3 q, Weighted_point_3 r, Weighted_point_3 s, FT w) ;
Comparison_result operator()(Weighted_point_3 p, Weighted_point_3 q, Weighted_point_3 r, FT w) ;
Comparison_result operator()(Weighted_point_3 p, Weighted_point_3 q, FT w) ;
Comparison_result operator()(Weighted_point_3 p, FT w) ;

Regular_triangulation_euclidean_traits_3<K,Weight>:: In_smallest_orthogonal_sphere_3

A predicate type. The operator() takes weighted points as arguments and returns the sign of the power distance of the last one with respect to the smallest sphere orthogonal to the others.
Sign operator()(Weighted_point_3 p, Weighted_point_3 q, Weighted_point_3 r, Weighted_point_3 s, Weighted_point_3 t) ;
Sign operator()(Weighted_point_3 p, Weighted_point_3 q, Weighted_point_3 r, Weighted_point_3 s) ;
Sign operator()(Weighted_point_3 p, Weighted_point_3 q, Weighted_point_3 r) ;
Sign operator()(Weighted_point_3 p, Weighted_point_3 q) ;

Regular_triangulation_euclidean_traits_3<K,Weight>:: Side_of_bounded_orthogonal_sphere_3

A predicate type. The operator() is similar to the operator() of *In_smallest_orthogonal_sphere_3* except that the returned type is not a *Sign* but belongs to the enum *Bounded_side* (A *NEGATIVE*, *ZERO* and *POSITIVE*) corresponding respectively to *ON_BOUNDED_SIDE*, *ON_BOUNDARY* and *ON_UNBOUNDED_SIDE*)).
Bounded_side operator() (Weighted_point_3 p, Weighted_point_3 q, Weighted_point_3 r, Weighted_point_3 s, Weighted_point_3 t) ;
Bounded_side operator() (Weighted_point_3 p, Weighted_point_3 q, Weighted_point_3 r, Weighted_point_3 s) ;
Bounded_side operator() (Weighted_point_3 p, Weighted_point_3 q, Weighted_point_3 r) ;

Regular_triangulation_euclidean_traits_3<K,Weight>:: Does_simplex_intersect_dual_support_3

A predicate type. The operator() takes weighted points as arguments, considers the subspace of points with equal power distance with respect to its arguments and the intersection of this subspace with the affine hull of the bare points associated to the arguments. The operator() returns *ON_BOUNDED_SIDE*, *ON_BOUNDARY* or *ON_UNBOUNDED_SIDE* according to the position of this intersection with respect to the simplex formed by the bare points. This predicate is useful for flow computations.
Bounded_side operator()(Weighted_point_3 p, Weighted_point_3 q, Weighted_point_3 r, Weighted_point_3 s) ;
Bounded_side operator()(Weighted_point_3 p, Weighted_point_3 q, Weighted_point_3 r) ;
Bounded_side operator()(Weighted_point_3 p, Weighted_point_3 q) ;

Types for constructor functors

Regular_triangulation_euclidean_traits_3<K,Weight>:: Construct_weighted_circumcenter_3

A constructor type. The operator() constructs the bare point which is the center of the smallest orthogonal sphere to the input weighted points.
Bare_point operator() (Weighted_point_3 p, Weighted_point_3 q, Weighted_point_3 r, Weighted_point_3 s);
Bare_point operator() (Weighted_point_3 p, Weighted_point_3 q, Weighted_point_3 r);
Bare_point operator() (Weighted_point_3 p, Weighted_point_3 q);

Regular_triangulation_euclidean_traits_3<K,Weight>:: Compute_power_product_3

A functor type. The operator() computes the power distance between its arguments.
FT operator() (Weighted_point_3 p, Weighted_point_3 q) ;

Regular_triangulation_euclidean_traits_3<K,Weight>:: Compute_squared_radius_smallest_orthogonal_sphere_3

A functor type. The operator() computes the squared radius of the smallest sphere orthogonal to the argument(s).
FT operator() (Weighted_point_3 p, Weighted_point_3 q, Weighted_point_3 r, Weighted_point_3 s);
FT operator() (Weighted_point_3 p, Weighted_point_3 q, Weighted_point_3 r);
FT operator() (Weighted_point_3 p, Weighted_point_3 q);
FT operator() (Weighted_point_3 p);

Regular_triangulation_euclidean_traits_3<K,Weight>:: Compute_critical_squared_radius_3

A functor type. The operator() takes weighted points as arguments and computes the squared radius of the sphere centered in the last point and orthogonal to the other weighted points. The last argument is a weighted point but its weight does not matter. This construction is ad hoc for pumping slivers. For robustness issue, a predicate to compare critical squared radii for a given last point should be needed.
FT operator() (Weighted_point_3 p, Weighted_point_3 q, Weighted_point_3 r, Weighted_point_3 s, Weighted_point_3 t);

Operations

The following functions give access to the predicate and constructor functors.

Power_test_3	*traits.power_test_3_object()*
Compare_power_distance_3	*traits.compare_power_distance_3_object()*
Compare_weighted_squared_radius_3	*traits.compare_weighted_squared_radius_3_object()*
In_smallest_orthogonal_sphere_3	*traits.in_smallest_orthogonal_sphere_3_object()*
Side_of_bounded_orthogonal_sphere_3	*traits.side_of_bounded_orthogonal_sphere_3_object()*
Does_simplex_intersect_dual_support_3	*traits.does_simplex_intersect_dual_support_3_object()*
Construct_weighted_circumcenter_3	*traits.construct_weighted_circumcenter_3_object()*
Compute_power_product_3	*traits.compute_power_product_3_object()*

Compute_squared_radius_smallest_orthogonal_sphere_3	
	traits.compute_squared_radius_smallest_orthogonal_sphere_3_object( )
Compute_critical_squared_radius_3	*traits.compute_critical_squared_radius_3_object()*

Concept

TriangulationCellBase_3

Definition

The cell base required by the basic triangulation does not need to store any geometric information, so only the requirements of the triangulation data structure apply.

However, for the Delaunay triangulation, the ability to store the circumcenter is provided (for optimization reasons), hence an additional requirement only in this case, and only when the dual functions are called.

Refines

TriangulationDSCellBase_3

DelaunayTriangulationTraits_3::Point_3

c.circumcenter(DelaunayTriangulationTraits_3 gt = DelaunayTriangulationTraits_3())

Returns the circumcenter.

Has Models

CGAL::Triangulation_cell_base_3
CGAL::Triangulation_cell_base_with_info_3
CGAL::Triangulation_cell_base_with_circumcenter_3

See Also

TriangulationVertexBase_3

TriangulationVertexBase_3

Concept

Definition

The vertex base used by the geometric triangulation must store a point. So we list here the additional requirements compared to a vertex base usable for the triangulation data structure.

Refines

TriangulationDSVertexBase_3

Types

TriangulationVertexBase_3:: Point	Must be the same as the point type *TriangulationTraits_3::Point_3* defined by the geometric traits class of the triangulation.

Creation

TriangulationVertexBase_3 v(Point p);	Constructs a vertex whose geometric embedding is point p.
TriangulationVertexBase_3 v(Point p, Cell_handle c);	Constructs a vertex embedding the point p and pointing to cell c.

Access Functions

Point	*v.point() const*	Returns the point.

Setting

void	*v.set_point(Point p)*	Sets the point.

I/O

istream&	*istream& is >> & v*	Inputs the non-combinatorial information given by the vertex: the point and other possible information.
ostream&	*ostream& os << v*	Outputs the non-combinatorial information given by the vertex: the point and other possible information.

Has Models

CGAL::Triangulation_vertex_base_3
CGAL::Triangulation_vertex_base_with_info_3

See Also

TriangulationCellBase_3

TriangulationVertexBaseWithInfo_3

Concept

Definition

A type model of this concept can be used as vertex base by a triangulation and provides an additional information storage.

Refines

TriangulationVertexBase_3

Types

TriangulationVertexBaseWithInfo_3:: Info	A type which is *DefaultConstructible* and *Assignable*.

Access Functions

Info	*v.info() const*	Returns a const reference to the object of type *Info* stored in the vertex.
Info&	*v.info()*	Returns a reference to the object of type *Info* stored in the vertex.

Has Models

CGAL::Triangulation_vertex_base_with_info_3

CGAL::Triangulation_simplex_3<Triangulation_3>

Definition

The class *Triangulation_simplex_3<Triangulation_3>* stores a simplex of any dimension defined by the *Triangulation_3* class. It also defines the operator less such that simplices can be stored in a *map* or a *set* of simplices. The simplex is invalidated by any change in the triangulation.

#include <CGAL/Triangulation_simplex_3.h>

Parameters

It is parameterized by the triangulation it derives the simplices from.

Types

typedef Triangulation_simplex_3<Triangulation_3>	*Simplex;*	The simplex class itself.
typedef Triangulation_3::Vertex_handle	*Vertex_handle;*	
typedef Triangulation_3::Edge	*Edge;*	
typedef Triangulation_3::Facet	*Facet;*	
typedef Triangulation_3::Cell_handle	*Cell_handle;*	
typedef Triangulation_3::Cell_circulator	*Cell_circulator;*	
typedef Triangulation_3::Facet_circulator	*Facet_circulator;*	
typedef Triangulation_3::Edge_iterator	*Edge_iterator;*	
typedef Triangulation_3::Facet_iterator	*Facet_iterator;*	
typedef Triangulation_3::Finite_vertices_iterator	*Finite_vertices_iterator;*	
typedef Triangulation_3::Finite_edges_iterator	*Finite_edges_iterator;*	
typedef Triangulation_3::Finite_facets_iterator	*Finite_facets_iterator;*	
typedef Triangulation_3::Finite_cells_iterator	*Finite_cells_iterator;*	

Creation

Triangulation_simplex_3<Triangulation_3>	*simplex*;	Initializes the simplex to an invalid simplex.
Triangulation_simplex_3<Triangulation_3>	*simplex(Vertex_handle vh);*	
Triangulation_simplex_3<Triangulation_3>	*simplex(Edge e);*	
Triangulation_simplex_3<Triangulation_3>	*simplex(Facet f);*	
Triangulation_simplex_3<Triangulation_3>	*simplex(Cell_handle ch);*	
Triangulation_simplex_3<Triangulation_3>	*simplex(Cell_circulator ccir);*	
Triangulation_simplex_3<Triangulation_3>	*simplex(Facet_circulator fcir);*	

Triangulation_simplex_3<Triangulation_3> simplex(Edge_iterator eit);
Triangulation_simplex_3<Triangulation_3> simplex(Facet_iterator fit);

Operations

int	*simplex.dimension() const*	
		returns the dimension of the simplex.
Vertex_handle	*Vertex_handle (simplex)*	Returns the *Vertex_handle* stored in the simplex. *Precondition*: dimension() == 0
Edge	*Edge (simplex)*	Returns the *Edge* stored in the simplex. *Precondition*: dimension() == 1
Facet	*Facet (simplex)*	Returns the *Facet* stored in the simplex. *Precondition*: dimension() == 2
Cell_handle	*Cell_handle (simplex)*	Returns the *Cell_handle* stored in the simplex. *Precondition*: dimension() == 3
Cell_handle	*simplex.incident_cell() const*	
		Returns a cell incident to the simplex.
bool	*simplex == s1*	Test whether two simplices are equal.
bool	*simplex < s1*	Defines a ordering on the simplices. This ordering depends on the memory layout and is independent of the geometry. Therefore, the ordering is not intrinsic

See Also

CGAL::Triangulation_3<TriangulationTraits_3,TriangulationDataStructure_3> . . . . page 2636

Concept

TriangulationHierarchyVertexBase_3

deprecated

Note : this concept is deprecated since CGAL 3.6, as the only class using it, *Triangulation_hierarchy_3* has been deprecated as well.

Definition

The vertex base used by *Triangulation_hierarchy_3* must provide access to two vertex handles for linking between the levels of the hierarchy.

Refines

TriangulationVertexBase_3

Access Functions

Vertex_handle	*v.up() const*	Returns the *Vertex_handle* pointing to the level above.
Vertex_handle	*v.down() const*	Returns the *Vertex_handle* pointing to the level below.

Setting

void	*v.set_up(Vertex_handle v)*	Sets the *Vertex_handle* pointing to the level above to *v*.
void	*v.set_down(Vertex_handle v)*	Sets the *Vertex_handle* pointing to the level below to *v*.

Has Models

CGAL::Triangulation_hierarchy_vertex_base_3

deprecated

RegularTriangulationCellBase_3

Concept

Definition

The regular triangulation of a set of weighted points does not necessarily have one vertex for each of the input points. Some of the input weighted points have no cell in the dual power diagrams and therefore do not correspond to a vertex of the regular triangulation. Those weighted points are said to be *hidden* points. A point which is hidden at a given time may appear later as a vertex of the regular triangulation upon removal on some other weighted point. Therefore, hidden points have to be stored somewhere. The regular triangulation stores those hidden points in its cells.

A hidden point can appear as vertex of the triangulation only when the three dimensional cell where its point component is located (the cell which hides it) is removed. Therefore we decided to store in each cell of a regular triangulation the list of hidden points that are located in the face. Thus points hidden by a face are easily reinserted in the triangulation when the face is removed.

The base cell of a regular triangulation has to be a model of the concept RegularTriangulationCellBase_3, which refines the concept *TriangulationCellBase_3* by adding in the cell a container to store hidden points.

Refines

TriangulationCellBase_3

Types

RegularTriangulationCellBase_3:: Point	Must be the same as the point type *TriangulationTraits_3::Point_3* defined by the geometric traits class of the triangulation.

Types

RegularTriangulationCellBase_3:: Point_iterator

Iterator of value type Point

Access Functions

Point_iterator *rcb.hidden_points_begin()*

Returns an iterator pointing to the first hidden point.

Point_iterator *rcb.hidden_points_end()*

Returns a past-the-end iterator.

Setting

void *rcb.hide_point(Point p)*

Adds *p* to the set of hidden points of the cell.

Has Models

CGAL::Regular_triangulation_cell_base_3

See Also

TriangulationCellBase_3

CGAL::Triangulation_cell_base_3<TriangulationTraits_3, TriangulationDSCellBase_3>

Definition

The class *Triangulation_cell_base_3* is a model of the concept *TriangulationCellBase_3*, the base cell of a 3D-triangulation.

This class can be used directly or can serve as a base to derive other classes with some additional attributes (a color for example) tuned for a specific application.

#include <CGAL/Triangulation_cell_base_3.h>

Parameters

The first template argument is the geometric traits class *TriangulationTraits_3*. It is actually not used by this class.

The second template argument is a combinatorial cell base class from which *Triangulation_cell_base_3* derives. It has the default value *Triangulation_ds_cell_base_3<>*.

Note that this model does not store the circumcenter, but computes it every time the circumcenter function is called. See *CGAL::Triangulation_cell_base_with_circumcenter_3* for a way to cache the circumcenter computation.

Is Model for the Concepts

TriangulationCellBase_3

Inherits From

TriangulationDSCellBase_3

See Also

CGAL::Triangulation_ds_cell_base_3
CGAL::Triangulation_cell_base_with_info_3
CGAL::Triangulation_cell_base_with_circumcenter_3
CGAL::Triangulation_vertex_base_3

CGAL::Triangulation_cell_base_with_info_3<Info, TriangulationTraits_3, TriangulationCellBase_3>

Definition

The class *Triangulation_cell_base_with_info_3* is a model of the concept *TriangulationCellBase_3*, the base cell of a 3D-triangulation. It provides an easy way to add some user defined information in cells. Note that input/output operators discard this additional information.

#include <CGAL/Triangulation_cell_base_with_info_3.h>

Parameters

The first template argument is the information the user would like to add to a cell. It has to be *DefaultConstructible* and *Assignable*.

The second template argument is the geometric traits class *TriangulationTraits_3*. It is actually not used by this class.

The third template argument is a cell base class from which *Triangulation_cell_base_with_info_3* derives. It has the default value *Triangulation_cell_base_3<TriangulationTraits_3>*.

Is Model for the Concepts

TriangulationCellBase_3

Inherits From

TriangulationCellBase_3

Types

typedef Info	*Info;*

Access Functions

const Info&	*v.info() const*	Returns a const reference to the object of type *Info* stored in the cell.
Info&	*v.info()*	Returns a reference to the object of type *Info* stored in the cell.

See Also

CGAL::Triangulation_cell_base_3
CGAL::Triangulation_cell_base_with_circumcenter_3 *CGAL::Triangulation_vertex_base_with_info_3*

CGAL::Triangulation_cell_base_with_circumcenter_3< DelaunayTriangulationTraits_3, TriangulationCellBase_3>

Class

Definition

The class *Triangulation_cell_base_with_circumcenter_3* is a model of the concept *TriangulationCellBase_3*, the base cell of a 3D-triangulation. It provides an easy way to cache the computation of the circumcenter of tetrahedra. Note that input/output operators discard this additional information.

All functions modifying the vertices of the cell, invalidate the cached circumcenter.

#include <CGAL/Triangulation_cell_base_with_circumcenter_3.h>

Parameters

The first template argument is the geometric traits class *DelaunayTriangulationTraits_3*.

The second template argument is a cell base class from which *Triangulation_cell_base_with_circumcenter_3* derives. It has the default value *Triangulation_cell_base_3<DelaunayTriangulationTraits_3>*.

Is Model for the Concepts

TriangulationCellBase_3

Inherits From

TriangulationCellBase_3

Access Functions

const DelaunayTriangulationTraits_3::Point_3&

v.circumcenter(const DelaunayTriangulationTraits_3&gt = DelaunayTriangulationTraits_3())

Computes the circumcenter of the tetrahedron, or retrieve it if already computed.

See Also

CGAL::Triangulation_cell_base_3
CGAL::Triangulation_cell_base_with_info_3

Class

CGAL::Triangulation_vertex_base_3<TriangulationTraits_3, TriangulationDSVertexBase_3>

Definition

The class *Triangulation_vertex_base_3* is a model of the concept *TriangulationVertexBase_3*, the base vertex of a 3D-triangulation. This class stores a point.

This class can be used directly or can serve as a base to derive other classes with some additional attributes (a color for example) tuned for a specific application.

#include <CGAL/Triangulation_vertex_base_3.h>

Parameters

The first template argument is the geometric traits class *TriangulationTraits_3* which provides the point type, *Point_3*. Users of the geometric triangulations (Section 40.2 and Chapter 39) are strongly advised to use the same geometric traits class *TriangulationTraits_3* as the one used for *Triangulation_3*. This way, the point type defined by the base vertex is the same as the point type defined by the geometric traits class.

The second template argument is a combinatorial vertex base class from which *Triangulation_vertex_base_3* derives. It has the default value *Triangulation_ds_vertex_base_3<>*.

Is Model for the Concepts

TriangulationVertexBase_3

Inherits From

TriangulationDSVertexBase_3

Types

typedef TriangulationTraits_3::Point_3	*Point;*

See Also

CGAL::Triangulation_cell_base_3
CGAL::Triangulation_ds_vertex_base_3
CGAL::Triangulation_vertex_base_with_info_3

CGAL::Triangulation_vertex_base_with_info_3<Info, TriangulationTraits_3, TriangulationVertexBase_3>

Class

Definition

The class *Triangulation_vertex_base_with_info_3* is a model of the concept *TriangulationVertexBase_3*, the base vertex of a 3D-triangulation. It provides an easy way to add some user defined information in vertices. Note that input/output operators discard this additional information.

#include <CGAL/Triangulation_vertex_base_with_info_3.h>

Parameters

The first template argument is the information the user would like to add to a vertex. It has to be *DefaultConstructible* and *Assignable*.

The second template argument is the geometric traits class *TriangulationTraits_3* which provides the *Point_3*.

The third template argument is a vertex base class from which *Triangulation_vertex_base_with_info_3* derives. It has the default value *Triangulation_vertex_base_3<TriangulationTraits_3>*.

Is Model for the Concepts

TriangulationVertexBase_3
TriangulationVertexBaseWithInfo_3

Inherits From

TriangulationVertexBase_3

Types

typedef Info	*Info;*

Access Functions

const Info&	*v.info() const*	Returns a const reference to the object of type *Info* stored in the vertex.
Info&	*v.info()*	Returns a reference to the object of type *Info* stored in the vertex.

See Also

CGAL::Triangulation_cell_base_with_info_3
CGAL::Triangulation_vertex_base_3

Class

CGAL::Triangulation_hierarchy_vertex_base_3<TriangulationVertexBase_3>

deprecated

Note : this class is deprecated since CGAL 3.6. Its functionality is now transparently added when using the *Fast_location* tag as the *LocationPolicy* template parameter in *Delaunay_triangulation_3*, instead of *Triangulation_hierarchy_3*.

Definition

This class is designed to be used as the vertex base class for *Triangulation_hierarchy_3*.

It inherits from its parameter *TriangulationVertexBase_3*, and adds the requirements in order to match the concept *TriangulationHierarchyVertexBase_3*, it does so by storing two *Vertex_handle*s. This design allows to use either a vertex base class provided by CGAL, or a user customized vertex base with additional functionalities.

#include <CGAL/Triangulation_hierarchy_vertex_base_3.h>

Parameters

It is parameterized by a model of the concept *TriangulationVertexBase_3*.

Is Model for the Concepts

TriangulationHierarchyVertexBase_3

Inherits From

TriangulationVertexBase_3

See Also

CGAL::Triangulation_hierarchy_3
CGAL::Triangulation_vertex_base_3
CGAL::Triangulation_vertex_base_with_info_3

deprecated

CGAL::Regular_triangulation_cell_base_3<Traits,Cb>

Definition

The class *Regular_triangulation_cell_base_3<Traits,Cb>* is a model of the concept *RegularTriangulationCellBase_3*. It is the default face base class of regular triangulations.

#include <CGAL/Regular_triangulation_cell_base_3.h>

Parameters

The template parameters *Traits* has to be a model of *RegularTriangulationTraits_3*.

The template parameter *Cb* has to be a model of *TriangulationCellBase_3*. By default, this parameter is instantiated by *CGAL::Triangulation_cell_base_3<Traits>*.

Is Model for the Concepts

RegularTriangulationCellBase_3

Inherits From

Cb

See Also

RegularTriangulationCellBase_3
RegularTriangulationTraits_3
CGAL::Regular_triangulation_3<Traits,Tds>

Enum

CGAL::Triangulation_3::Locate_type

Definition

The enum *Locate_type* is defined by *Triangulation_3* to specify which case occurs when locating a point in the triangulation.

enum Locate_type { *VERTEX=0,*
EDGE,
FACET,
CELL,
OUTSIDE_CONVEX_HULL,
OUTSIDE_AFFINE_HULL}

See Also

CGAL::Triangulation_3

WeightedPoint

Definition

The concept WeightedPoint is needed by *Regular_triangulation_euclidean_traits_3*. It must fulfill the following requirements:

Types

WeightedPoint:: Point	The point type
WeightedPoint:: Weight	The weight type
typedef Point::RT RT;	The ring type

Creation

WeightedPoint wp(Point p=Point(), Weight w = Weight(0));

Access Functions

Point	*point() const*
Weight	*weight() const*

Has Models

Weighted_point.

See Also

CGAL::Regular_triangulation_euclidean_traits_3
CGAL::Regular_triangulation_3.

Class

CGAL::Regular_triangulation_filtered_traits_3<FK>

deprecated

Note : this class is deprecated since CGAL 3.6. The class *CGAL::Regular_triangulation_euclidean_traits_3* should be used instead. Filtered predicates are automatically used if the boolean *Has_filtered_predicates* in the kernel provided as template parameter of that class is set to *true*.

Definition

The class *Regular_triangulation_filtered_traits_3<FK>* is designed as a traits class for the class *Regular_triangulation_3<RegularTriangulationTraits_3,TriangulationDataStructure_3>*. Its difference with *Regular_triangulation_euclidean_traits_3* is that it provides filtered predicates which are meant to be fast and exact.

The first argument *FK* must be a model of the *Kernel* concept, and it is also restricted to be a instance of the *Filtered_kernel* template.

#include <CGAL/Regular_triangulation_filtered_traits_3.h>

Is Model for the Concepts

RegularTriangulationTraits_3

Inherits From

Regular_triangulation_euclidean_traits_3<FK>

See Also

CGAL::Regular_triangulation_euclidean_traits_3.

deprecated

Chapter 40

3D Triangulation Data Structure

Sylvain Pion and Monique Teillaud

Contents

A geometric triangulation has two aspects: the combinatorial structure, which gives the incidence and adjacency relations between faces, and the geometric information related to the position of vertices.

CGAL provides 3D geometric triangulations in which these two aspects are clearly separated. As described in Chapter 39, a geometric triangulation of a set of points in $\mathbb{R}^d$, $d \leq 3$ is a partition of the whole space $\mathbb{R}^d$ into cells having $d+1$ vertices. Some of them are infinite, they are obtained by linking an additional vertex at infinity to each facet of the convex hull of the points (see Section 39.1). The underlying combinatorial graph of such a triangulation without boundary of $\mathbb{R}^d$ can be seen as a triangulation of the topological sphere S^d in $\mathbb{R}^{d+1}$.

This chapter deals with 3D-triangulation data structures, meant to maintain the combinatorial information for 3D-geometric triangulations. The reader interested in geometric triangulations of $\mathbb{R}^3$ is advised to read Chapter 39.

40.1 Representation

In CGAL, a 3D triangulation data structure is a container of cells (3-faces) and vertices (0-faces).

Following the standard vocabulary of simplicial complexes, an i-face f_i and a j-face f_j $(0 \leq j < i \leq 3)$ are said to be *incident* in the triangulation if f_j is a (sub)face of f_i, and two i-faces $(0 \leq i \leq 3)$ are said to be *adjacent* if they share a commun incident (sub)face.

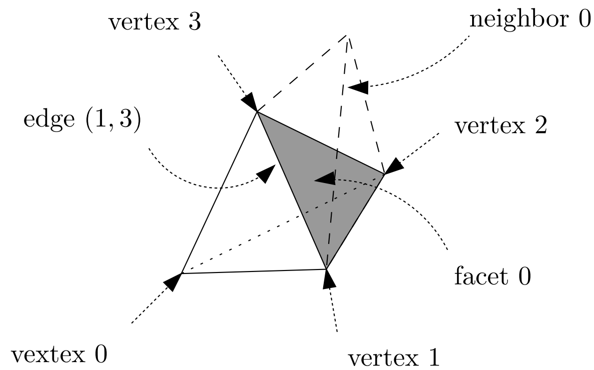


Figure 40.1: Representation.

Each cell gives access to its four incident vertices and to its four adjacent cells. Each vertex gives direct access to one of its incident cells, which is sufficient to retrieve all the incident cells when needed.

The four vertices of a cell are indexed with 0, 1, 2 and 3. The neighbors of a cell are also indexed with 0, 1, 2, 3 in such a way that the neighbor indexed by i is opposite to the vertex with the same index (see Figure 40.1).

Edges (1-faces) and facets (2-faces) are not explicitly represented: a facet is given by a cell and an index (the facet i of a cell c is the facet of c that is opposite to the vertex of index i) and an edge is given by a cell and two indices (the edge *(i,j)* of a cell c is the edge whose endpoints are the vertices of indices i and j of c).

Degenerate Dimensions As CGAL explicitly deals with all degenerate cases, a 3D-triangulation data structure in CGAL can handle the cases when the dimension of the triangulation is lower than 3.

Thus, a 3D-triangulation data structure can store a triangulation of a topological sphere S^d of $\mathbb{R}^{d+1}$, for any $d \in \{-1, 0, 1, 2, 3\}$.

Let us give, for each dimension, the example corresponding to the triangulation data structure having a minimal number of vertices, i.e. a simplex. These examples are illustrated by presenting their usual geometric embedding.

- *dimension 3.* The triangulation data structure consists of the boundary of a 4-dimensional simplex, which has 5 vertices. A geometric embedding consists in choosing one of these vertices to be infinite, thus four of the five 3-cells become infinite: the geometric triangulation has one finite tetrahedron remaining, each of its facets being incident to an infinite cell. See Figure 40.2.
- *dimension 2.* We have 4 vertices forming one 3-dimensional simplex, i.e. the boundary of a tetrahedron. The geometric embedding in the plane results from choosing one of these vertices to be infinite, then the geometric triangulation has one finite triangle whose edges are incident to the infinite triangles. See Figure 40.3.
- *dimension 1.* A 2-dimensional simplex (a triangle) has 3 vertices. The geometric embedding is an edge whose vertices are linked to an infinite point. See Figure 40.4.

The last three cases are defined uniquely:

- *dimension 0.* A 0-dimensional triangulation is combinatorially equivalent to the boundary of a 1-dimensional simplex (an edge), which consists of 2 vertices. One of them becomes infinite in the geometric embedding, and there is only one finite vertex remaining. The two vertices are adjacent.

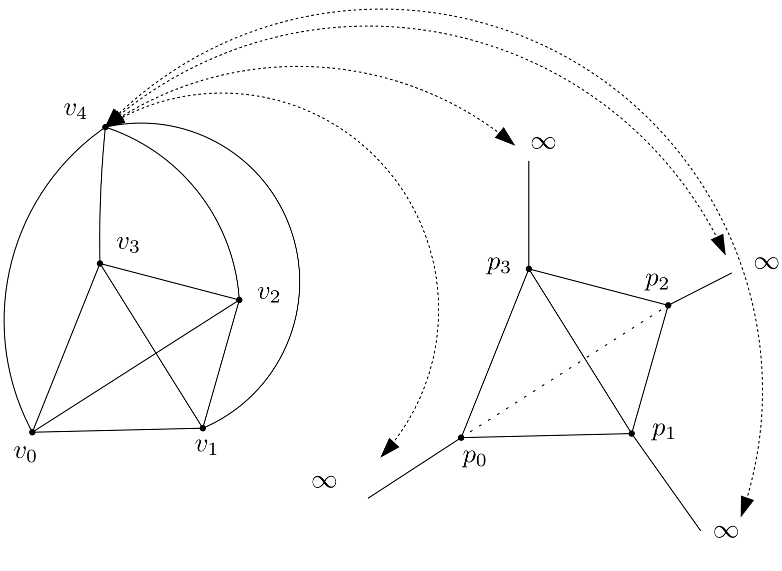


Figure 40.2: 4D simplex and a 3D geometric embedding.

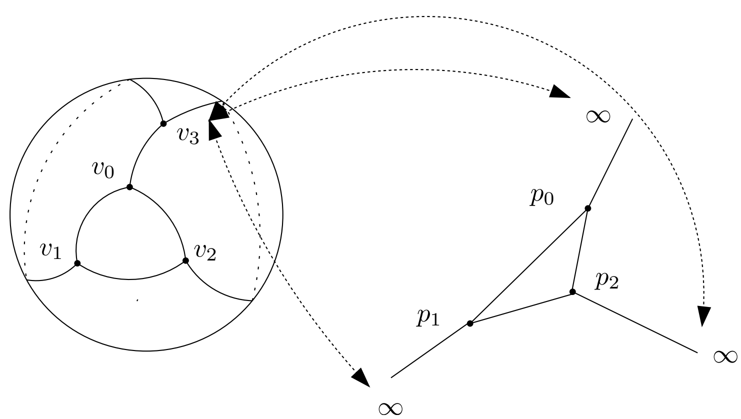


Figure 40.3: 3D simplex and a 2D geometric embedding.

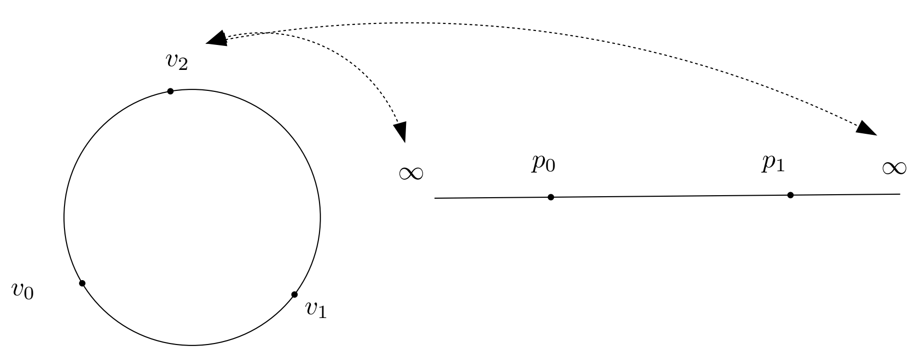


Figure 40.4: 2D simplex and a 1D geometric embedding.

- *dimension -1.* This dimension is a convention to represent a 0-dimensional simplex, that is a sole vertex, which will be geometrically embedded as an "empty" triangulation, having only one infinite vertex.
- *dimension -2.* This is also a convention. The triangulation data structure has no vertex. There is no associated geometric triangulation.

Note that the notion of infinite vertex has no meaning for the triangulation data structure. The infinite vertex of the geometric embedding is a vertex that cannot be distinguished from the other vertices in the combinatorial triangulation.

The same cell class is used in all cases: triangular faces in 2D can be considered as degenerate cells, having only three vertices (resp. neighbors) numbered $(0,1,2)$; edges in 1D have only two vertices (resp. neighbors) numbered 0 and 1.

The implicit representation of facets (resp. edges) still holds for degenerate (< 3) dimensions : in dimension 2, each cell has only one facet of index 3, and 3 edges $(0,1)$, $(1,2)$ and $(2,0)$; in dimension 1, each cell has one edge $(0,1)$.

Validity A 3D combinatorial triangulation is said to be *locally valid* iff the following is true:

(a) When a cell c has a neighbor pointer to another cell c', then reciprocally this cell c' has a neighbor pointer to c, and c and c' have three vertices in common. These cells are called adjacent.

(b) The cells have a coherent orientation: if two cells c_1 and c_2 are adjacent and share a facet with vertices u,v,w, then the vertices of c_1 are numbered $(v_0^1 = u, v_1^1 = v, v_2^1 = w, v_3^1)$, and the vertices of c_2 are numbered $(v_0^2 = v, v_1^2 = u, v_2^2 = w, v_3^2)$, up to positive permutations of $(0,1,2,3)$. In other words, if we embed the triangulation in $\mathbb{R}^3$, then the fourth vertices v_3^1 and v_3^2 of c_1 and c_2 see the common facet in opposite orientations. See Figure 40.5.

The set σ_4 of permutations of $(0,1,2,3)$ has cardinality 24, and the set of positive permutations A_4 has cardinality 12. Thus, for a given orientation, there are up to 12 different orderings of the four vertices of a cell. Note that cyclic permutations are negative and so do not preserve the orientation of a cell.

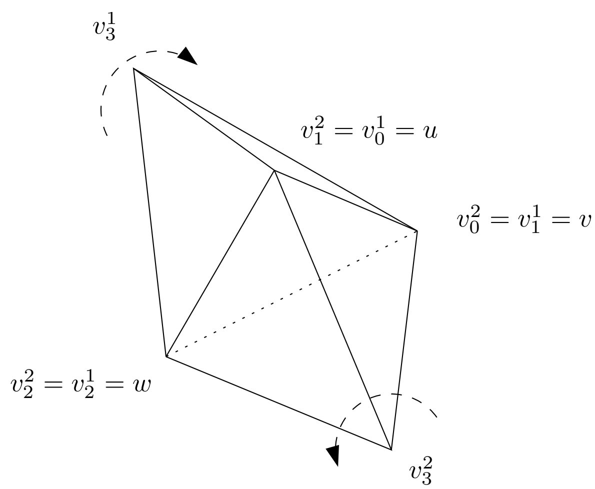


Figure 40.5: Coherent orientations of two cells (3-dimensional case).

The *is_valid()* method provided by *Triangulation_data_structure_3* checks the local validity of a given triangulation data structure.

40.2 Software Design

The 3D-triangulation data structure class of CGAL, *Triangulation_data_structure_3*, is designed to be used as a combinatorial layer upon which a geometric layer can be built [Ket98]. This geometric layer is typically one of the 3D-triangulation classes of CGAL: *Triangulation_3*, *Delaunay_triangulation_3* and *Regular_triangulation_3*. This relation is described in more details in Chapter 39, where the Section 39.4 explains other important parts of the design related to the geometry.

We focus here on the design of the triangulation data structure (TDS) itself, which the Figure 40.6 illustrates.

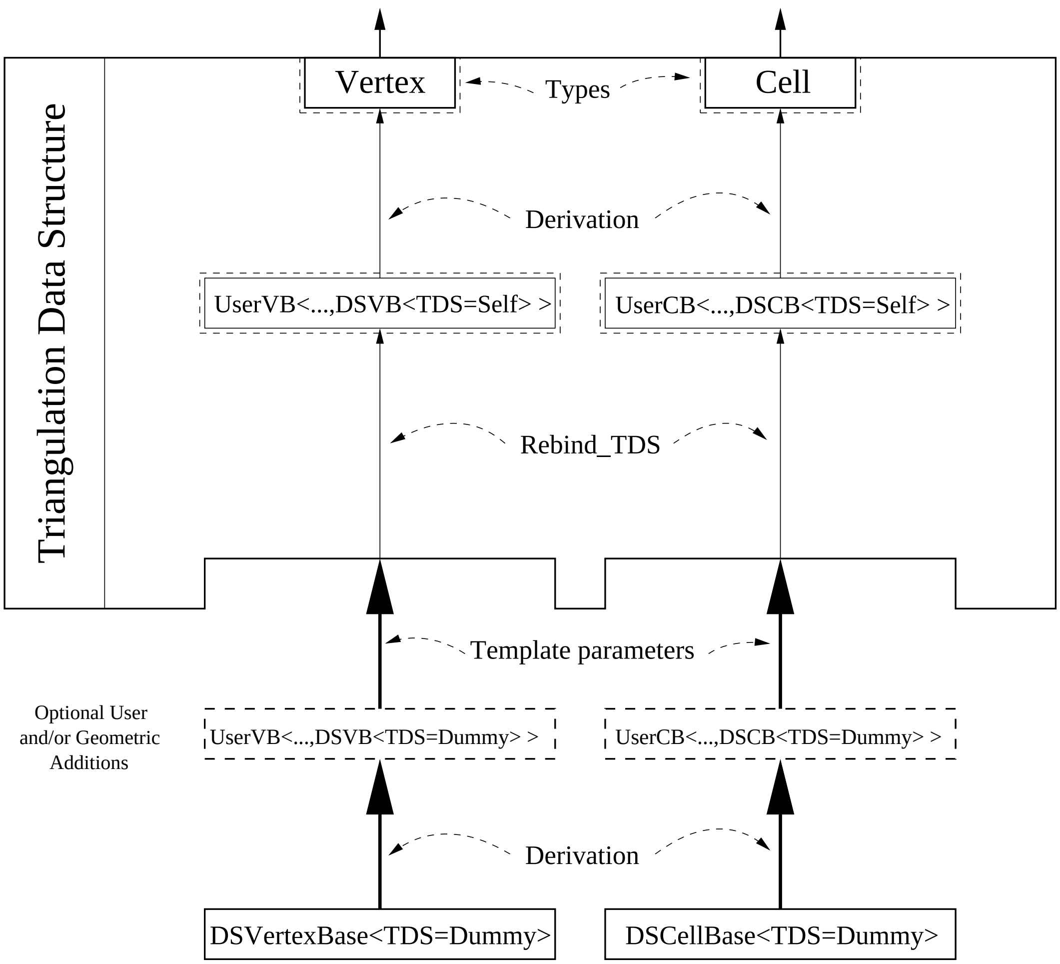


Figure 40.6: Triangulation Data Structure software design.

40.2.1 Flexibility of the Design

In order for the user to be able to add his own data in the vertices and cells, the design of the TDS is split into two layers:

- In the bottom layer, the (vertex and cell) base classes store elementary incidence and adjacency (and possibly geometric or other) information. These classes are parameterized by the TDS which provides the handle types. (They can also be parameterized by a geometric traits class or anything else.) A vertex stores a *Cell_handle*, and a cell stores four *Vertex_handle*s and four *Cell_handle*s.

- The middle layer is the TDS, which is purely combinatorial. It provides operations such as insertion of a new vertex in a given cell, on a 1 or 2-face. It also allows one, if the dimension of the triangulation is smaller than 3, to insert a vertex so that the dimension of the triangulation is increased by one. The TDS is responsible for the combinatorial integrity of the eventual geometric triangulation built on top of it (the upper layer, see Chapter 39).

The user has several ways to add his own data in the vertex and cell base classes used by the TDS. He can either:

- use the classes *Triangulation_vertex_base_with_info_3* and *Triangulation_cell_base_with_info_3*, which allow to add one data member of a user provided type, and give access to it.
- derive his own classes from the default base classes *Triangulation_ds_vertex_base*, and *Triangulation_ds_cell_base* (or the geometric versions typically used by the geometric layer, *Triangulation_vertex_base*, and *Triangulation_cell_base*).
- write his own base classes following the requirements given by the concepts *TriangulationCellBase_3* and *TriangulationVertexBase_3* (described in page 2684 and page 2685).

40.2.2 Cyclic Dependency

Since adjacency relations are stored in the vertices and cells, it means that the vertex and cell base classes have to be able to store handles (an entity akin to pointers) to their neighbors in the TDS. This in turns means that the vertex and cell base classes have to know the types of these handles, which are provided by the TDS. So in a sense, the base classes are parameterized by the TDS, and the TDS is parameterized by the vertex and cell base classes ! This is a cycle which cannot be resolved easily.

The solution that we have chosen is similar to the mechanism used by the standard class *std::allocator*: the vertex and cell base classes are initially given a fake or dummy TDS template parameter, whose unique purpose is to provide the types that can be used by the vertex and cell base classes (such as handles). Then, inside the TDS itself, these base classes are *rebound* to the real TDS type, that is we obtain the same vertex and cell base classes, but parameterized with the real TDS instead of the dummy one. Rebinding is performed by a nested template class of the vertex and cell base classes (see code below), which provides a type which is the rebound vertex or cell base class[1].

Here is how it works, schematically:

```
template < class Vb, class Cb >
class TDS
{
  typedef TDS<Vb, Cb>    Self;

  // Rebind the vertex and cell base to the actual TDS (Self).
  typedef typename Vb::template Rebind_TDS<Self>::Other  VertexBase;
  typedef typename Cb::template Rebind_TDS<Self>::Other  CellBase;

  // ... further internal machinery leads to the final public types:
public:
  typedef ...  Vertex;
  typedef ...  Cell;
```

[1]It is logically equivalent to a mechanism that does not exist yet in the C++ language: *template typedef* or *template aliasing*

```
  typedef ...  Vertex_handle;
  typedef ...  Cell_handle;
};

template < class TDS = ... >  // The default is some internal type faking
     a TDS
class Triangulation_ds_vertex_base_3
{
public:
  template < class TDS2 >
  struct Rebind_TDS {
    typedef Triangulation_ds_vertex_base_3<TDS2>    Other;
  };
...
};
```

When derivation is used for the vertex or cell base classes, which is the case at the geometric level with *Triangulation_vertex_base_3*, then it gets slightly more involved because its base class has to be rebound as well:

```
template < class GT, class Vb = Triangulation_ds_vertex_base_3<> >
class Triangulation_vertex_base_3 : public Vb
{
public:
  template < class TDS2 >
  struct Rebind_TDS {
    typedef typename Vb::template Rebind_TDS<TDS2>::Other  Vb2;
    typedef Triangulation_vertex_base_3<GT, Vb2>           Other;
  };
...
};
```

40.3 Examples

40.3.1 Incremental Construction

The following example shows how to construct a 3D triangulation data structure by inserting vertices.

```
#include <CGAL/Triangulation_data_structure_3.h>
#include <iostream>
#include <fstream>
#include <cassert>
#include <vector>

typedef CGAL::Triangulation_data_structure_3<>      Tds;

typedef Tds::size_type                              size_type;
typedef Tds::Cell_handle                            Cell_handle;
typedef Tds::Vertex_handle                          Vertex_handle;

int main()
```

```
{
  Tds T;

  assert( T.number_of_vertices() == 0 );
  assert( T.dimension() == -2 );
  assert( T.is_valid() );

  std::vector<Vertex_handle> PV(7);

  PV[0] = T.insert_increase_dimension();
  assert( T.number_of_vertices() == 1 );
  assert( T.dimension() == -1 );
  assert( T.is_valid() );

  // each of the following insertions of vertices increases the dimension
  for ( int i=1; i<5; i++ ) {
    PV[i] = T.insert_increase_dimension(PV[0]);
    assert( T.number_of_vertices() == (size_type) i+1 );
    assert( T.dimension() == i-1 );
    assert( T.is_valid() );
  }
  assert( T.number_of_cells() == 5 );

  // we now have a simplex in dimension 4

  // cell incident to PV[0]
  Cell_handle c = PV[0]->cell();
  int ind;
  bool check = c->has_vertex( PV[0], ind );
  assert( check );
  // PV[0] is the vertex of index ind in c

  // insertion of a new vertex in the facet opposite to PV[0]
  PV[5] = T.insert_in_facet(c, ind);

  assert( T.number_of_vertices() == 6 );
  assert( T.dimension() == 3 );
  assert( T.is_valid() );

  // insertion of a new vertex in c
  PV[6] = T.insert_in_cell(c);

  assert( T.number_of_vertices() == 7 );
  assert( T.dimension() == 3 );
  assert( T.is_valid() );

  std::ofstream oFileT("output_tds",std::ios::out);
  // writing file output_tds;
  oFileT << T;

  return 0;
}
```

File: examples/Triangulation_3/tds.cpp

40.3.2 Cross-Linking Between a 2D and a 3D Data Structures

This example program illustrates how to setup a 2D and a 3D triangulation data structures whose vertices respectively store vertex handles of the other one.

```
#include <CGAL/Triangulation_data_structure_2.h>
#include <CGAL/Triangulation_data_structure_3.h>
#include <cassert>

// declare the 2D vertex base type, parametrized by some 3D TDS.
template < typename T3, typename Vb =
    CGAL::Triangulation_ds_vertex_base_2<> >
class My_vertex_2;

// declare the 3D vertex base type, parametrized by some 2D TDS.
template < typename T2, typename Vb =
    CGAL::Triangulation_ds_vertex_base_3<> >
class My_vertex_3;

// Then, we have to break the dependency cycle.

// we need to refer to a dummy 3D TDS.
typedef
    CGAL::Triangulation_ds_vertex_base_3<>::Triangulation_data_structure
        Dummy_tds_3;
// the 2D TDS, initially plugging a dummy 3D TDS in the vertex type
// (to break the dependency cycle).
typedef CGAL::Triangulation_data_structure_2<My_vertex_2<Dummy_tds_3> >
    TDS_2;
// the 3D TDS, here we can plug the 2D TDS directly.
typedef CGAL::Triangulation_data_structure_3<My_vertex_3<TDS_2> >
    TDS_3;


template < typename T3, typename Vb >
class My_vertex_2
  : public Vb
{
public:
  typedef typename Vb::Face_handle    Face_handle;

  template <typename TDS2>
  struct Rebind_TDS {
    typedef typename Vb::template Rebind_TDS<TDS2>::Other  Vb2;
    // we also have to break the cycle here by hardcoding TDS_3 instead
   of T3.
    typedef My_vertex_2<TDS_3, Vb2>                           Other;
  };

  My_vertex_2() {}
```

```
  My_vertex_2(Face_handle f) : Vb(f) {}

  // we store a vertex handle of the 3D TDS.
  typename T3::Vertex_handle v3;
};

template < typename T2, typename Vb >
class My_vertex_3
  : public Vb
{
public:
  typedef typename Vb::Cell_handle    Cell_handle;

  template <typename TDS2>
  struct Rebind_TDS {
    typedef typename Vb::template Rebind_TDS<TDS2>::Other  Vb2;
    typedef My_vertex_3<T2, Vb2>                           Other;
  };

  My_vertex_3() {}

  My_vertex_3(Cell_handle c) : Vb(c) {}

  // we store a vertex handle of the 2D TDS.
  typename T2::Vertex_handle v2;
};


int main() {
  TDS_2 t2;
  TDS_3 t3;

  TDS_2::Vertex_handle v2 = t2.insert_dim_up();
  TDS_3::Vertex_handle v3 = t3.insert_increase_dimension();

  v2->v3 = v3;
  v3->v2 = v2;

  assert(t2.is_valid());
  assert(t3.is_valid());
  return 0;
}
```

File: examples/Triangulation_3/linking_2d_and_3d.cpp

40.4 Design and Implementation History

Monique Teillaud introduced the triangulation of the topological sphere S^d in $\mathbb{R}^{d+1}$ to manage the underlying graph of geometric triangulations and handle degenerate dimensions [Tei99].

Sylvain Pion improved the software in several ways, in particular regarding the memory management.

3D Triangulation Data Structure Reference Manual

Sylvain Pion and Monique Teillaud

The triangulation data structure is able to represent a triangulation of a topological sphere S^d of $\mathbb{R}^{d+1}$, for $d \in \{-1,0,1,2,3\}$. (See 40.1.)

The vertex class of a 3D-triangulation data structure must define a number of types and operations. The requirements that are of geometric nature are required only when the triangulation data structure is used as a layer for the geometric triangulation classes. (See Section 40.2.)

The cell class of a triangulation data structure stores four handles to its four vertices and four handles to its four neighbors. The vertices are indexed 0, 1, 2, and 3 in a consistent order. The neighbor indexed i lies opposite to vertex i.

In degenerate dimensions, cells are used to store faces of maximal dimension: in dimension 2, each cell represents only one facet of index 3, and 3 edges $(0,1)$, $(1,2)$ and $(2,0)$; in dimension 1, each cell represents one edge $(0,1)$. (See Section 40.1.)

40.5 Classified Reference Pages

Concepts

Classes

This class is a model for the concept of the 3D-triangulation data structure *TriangulationDataStructure_3*. It is templated by base classes for vertices and cells.

CGAL provides base vertex classes and base cell classes:

Helper Classes

It defines operations on the indices of vertices and neighbors within a cell of a triangulation.

40.6 Alphabetical List of Reference Pages

TriangulationDataStructure_3

Concept

Definition

3D-triangulation data structures are meant to maintain the combinatorial information for 3D-geometric triangulations.

In CGAL, a triangulation data structure is a container of cells (3-faces) and vertices (0-faces). Following the standard vocabulary of simplicial complexes, an i-face f_i and a j-face f_j $(0 \leq j < i \leq 3)$ are said to be *incident* in the triangulation if f_j is a (sub)face of f_i, and two i-faces $(0 \leq i \leq 3)$ are said to be *adjacent* if they share a common incident (sub)face.

Each cell gives access to its four incident vertices and to its four adjacent cells. Each vertex gives direct access to one of its incident cells, which is sufficient to retrieve all the incident cells when needed.

The four vertices of a cell are indexed with 0, 1, 2 and 3. The neighbors of a cell are also indexed with 0, 1, 2, 3 in such a way that the neighbor indexed by i is opposite to the vertex with the same index (see Figure 40.1).

Edges (1-faces) and facets (2-faces) are not explicitly represented: a facet is given by a cell and an index (the facet i of a cell c is the facet of c that is opposite to the vertex of index i) and an edge is given by a cell and two indices (the edge *(i,j)* of a cell c is the edge whose endpoints are the vertices of indices i and j of c).

As CGAL explicitly deals with all degenerate cases, a 3D-triangulation data structure in CGAL can handle the cases when the dimension of the triangulation is lower than 3 (see Section 40.1).

Thus, a 3D-triangulation data structure can store a triangulation of a topological sphere S^d of $\mathbb{R}^{d+1}$, for any $d \in \{-1,0,1,2,3\}$.

The second template parameter of the basic triangulation class (see Chapter 39 , page 2636) *Triangulation_3* is a triangulation data structure class. (See Chapter 40.)

To ensure all the **flexibility** of the class *Triangulation_3*, a model of a triangulation data structure must be templated by the base vertex and the base cell classes (see 40.1): *TriangulationDataStructure_3< TriangulationVertexBase_3,TriangulationCellBase_3>*. The optional functionalities related to geometry are compulsory for this use as a template parameter of *Triangulation_3*.

A class that satisfies the requirements for a triangulation data structure class must provide the following types and operations.

Types

TriangulationDataStructure_3:: Vertex	Vertex type
TriangulationDataStructure_3:: Cell	Cell type
TriangulationDataStructure_3:: size_type	Size type (unsigned integral type)
TriangulationDataStructure_3:: difference_type	Difference type (signed integral type)

Vertices and cells are usually manipulated via *handles*, which support the two dereference operators *operator** and *operator->*.

TriangulationDataStructure_3:: Vertex_handle
TriangulationDataStructure_3:: Cell_handle

Requirements for *Vertex* and *Cell* are described in *TriangulationDataStructure_3::Vertex* and *TriangulationDataStructure_3::Cell* (page 2732 and page 2730).

advanced

TriangulationDataStructure_3:: template <typename Vb2> struct Rebind_vertex

This nested template class allows to get the type of a triangulation data structure that only changes the vertex type. It has to define a type *Other* which is a *rebound* triangulation data structure, that is, the one whose *TriangulationDSVertexBase_3* will be *Vb2*.

TriangulationDataStructure_3:: template <typename Cb2> struct Rebind_cell

This nested template class allows to get the type of a triangulation data structure that only changes the cell type. It has to define a type *Other* which is a *rebound* triangulation data structure, that is, the one whose *TriangulationDSCellBase_3* will be *Cb2*.

advanced

typedef Triple<Cell_handle, int, int>	*Edge;*	*(c,i,j)* is the edge of cell *c* whose vertices indices are *i* and *j*. (See Section 40.1.)
typedef std::pair<Cell_handle, int>	*Facet;*	*(c,i)* is the facet of *c* opposite to the vertex of index *i*. (See Section 40.1.)

The following iterators allow one to visit all the vertices, edges, facets and cells of the triangulation data structure. They are all bidirectional, non-mutable iterators.

TriangulationDataStructure_3:: Cell_iterator
TriangulationDataStructure_3:: Facet_iterator
TriangulationDataStructure_3:: Edge_iterator
TriangulationDataStructure_3:: Vertex_iterator

The following circulators allow us to visit all the cells and facets incident to a given edge. They are bidirectional and non-mutable.

TriangulationDataStructure_3:: Facet_circulator
TriangulationDataStructure_3:: Cell_circulator

Iterators and circulators are convertible to the corresponding handles, thus the user can pass them directly as arguments to the functions.

Creation

TriangulationDataStructure_3 tds; Default constructor.

TriangulationDataStructure_3	*tds(tds1);*	Copy constructor. All vertices and cells are duplicated.
TriangulationDataStructure_3&	*tds = tds1*	Assignment operator. All vertices and cells are duplicated, and the former data structure of *tds* is deleted.

Vertex_handle *tds.copy_tds(tds1, Vertex_handle v = Vertex_handle())*

tds1 is copied into *tds*. If *v != Vertex_handle()*, the vertex of *tds* corresponding to *v* is returned, otherwise *Vertex_handle()* is returned.
Precondition: The optional argument *v* is a vertex of *tds1*.

void	*tds.swap(& tds1)*	Swaps *tds* and *tds1*. There is no copy of cells and vertices, thus this method runs in constant time. This method should be preferred to *tds*=*tds1* or *tds*(*tds1*) when *tds1* is deleted after that.
void	*tds.clear()*	Deletes all cells and vertices. *tds* is reset as a triangulation data structure constructed by the default constructor.

Operations

Access Functions

int	*tds.dimension() const*	The dimension of the triangulated topological sphere.
size_type	*tds.number_of_vertices() const*	The number of vertices. Note that the triangulation data structure has one more vertex than an associated geometric triangulation, if there is one, since the infinite vertex is a standard vertex and is thus also counted.
size_type	*tds.number_of_cells() const*	The number of cells. Returns 0 if *tds.dimension()*<3.

Non constant-time access functions

size_type	*tds.number_of_facets() const*	The number of facets. Returns 0 if *tds.dimension()*<2.
size_type	*tds.number_of_edges() const*	The number of edges. Returns 0 if *tds.dimension()*<1.

advanced

Setting

void	*tds.set_dimension(int n)*	Sets the dimension to *n*.

advanced

Queries

bool *tds.is_vertex(Vertex_handle v) const*

Tests whether *v* is a vertex of *tds*.

bool *tds.is_edge(Cell_handle c, int i, int j) const*

Tests whether *(c,i,j)* is an edge of *tds*. Answers *false* when *dimension()* < 1 .
Precondition: $i, j \in \{0,1,2,3\}$

bool *tds.is_edge(Vertex_handle u, Vertex_handle v, Cell_handle & c, int & i, int & j) const*

Tests whether *(u,v)* is an edge of *tds*. If the edge is found, it computes a cell *c* having this edge and the indices *i* and *j* of the vertices *u* and *v*, in this order.

bool *tds.is_edge(Vertex_handle u, Vertex_handle v) const*

Tests whether *(u,v)* is an edge of *tds*.

bool *tds.is_facet(Cell_handle c, int i) const*

Tests whether *(c,i)* is a facet of *tds*. Answers *false* when *dimension()* < 2 .
Precondition: $i \in \{0,1,2,3\}$

bool *tds.is_facet(Vertex_handle u, Vertex_handle v, Vertex_handle w, Cell_handle & c, int & i, int & j, int & k) const*

Tests whether *(u,v,w)* is a facet of *tds*. If the facet is found, it computes a cell *c* having this facet and the indices *i*, *j* and *k* of the vertices *u*, *v* and *w*, in this order.

bool *tds.is_cell(Cell_handle c) const* Tests whether *c* is a cell of *tds*. Answers *false* when *dimension()* < 3 .

bool *tds.is_cell(Vertex_handle u, Vertex_handle v, Vertex_handle w, Vertex_handle t, Cell_handle & c, int & i, int & j, int & k, int & l) const*

Tests whether *(u,v,w,t)* is a cell of *tds*. If the cell *c* is found, it computes the indices *i*, *j*, *k* and *l* of the vertices *u*, *v*, *w* and *t* in *c*, in this order.

There is a method *has_vertex* in the cell class. The analogous methods for facets are defined here.

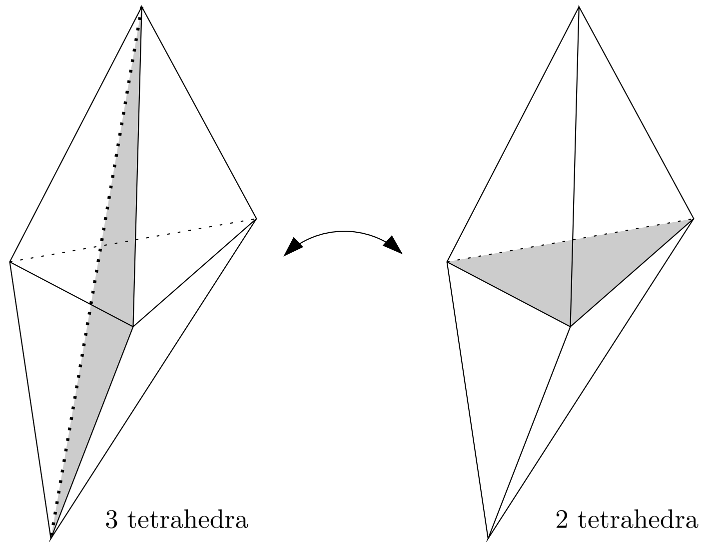


Figure 40.7: Flips.

bool *tds.has_vertex(Facet f, Vertex_handle v, int & j) const*

If v is a vertex of f, then j is the index of v in the cell *f.first*, and the method returns *true*.
Precondition: *tds*.dimension()=3

bool *tds.has_vertex(Cell_handle c, int i, Vertex_handle v, int & j) const*

Same for facet *(c,i)*. Computes the index j of v in c.

bool *tds.has_vertex(Facet f, Vertex_handle v) const*
bool *tds.has_vertex(Cell_handle c, int i, Vertex_handle v) const*

Same as the first two methods, but these two methods do not return the index of the vertex.

The following three methods test whether two facets have the same vertices.

bool *tds.are_equal(Facet f, Facet g) const*
bool *tds.are_equal(Cell_handle c, int i, Cell_handle n, int j) const*
bool *tds.are_equal(Facet f, Cell_handle n, int j) const*

For these three methods:
Precondition: *tds*.dimension()=3.

Flips

Two kinds of flips exist for a three-dimensional triangulation. They are reciprocal. To be flipped, an edge must be incident to three tetrahedra. During the flip, these three tetrahedra disappear and two tetrahedra appear. Figure 40.7(left) shows the edge that is flipped as bold dashed, and one of its three incident facets is shaded. On the right, the facet shared by the two new tetrahedra is shaded.

The following methods guarantee the validity of the resulting 3D combinatorial triangulation. Moreover the flip operations do not invalidate the vertex handles, and only invalidate the cell handles of the affected cells.

Flips for a 2d triangulation are not implemented yet

bool *tds.flip(Edge e)*
bool *tds.flip(Cell_handle c, int i, int j)*

Before flipping, these methods check that edge *e=(c,i,j)* is flippable (which is quite expensive). They return *false* or *true* according to this test.

void *tds.flip_flippable(Edge e)*
void *tds.flip_flippable(Cell_handle c, int i, int j)*

Should be preferred to the previous methods when the edge is known to be flippable.
Precondition: The edge is flippable.

bool *tds.flip(Facet f)*
bool *tds.flip(Cell_handle c, int i)*

Before flipping, these methods check that facet *f=(c,i)* is flippable (which is quite expensive). They return *false* or *true* according to this test.

void *tds.flip_flippable(Facet f)*
void *tds.flip_flippable(Cell_handle c, int i)*

Should be preferred to the previous methods when the facet is known to be flippable.
Precondition: The facet is flippable.

Insertions

The following modifier member functions guarantee the combinatorial validity of the resulting triangulation.

Vertex_handle *tds.insert_in_cell(Cell_handle c)*

Creates a new vertex, inserts it in cell c and returns its handle. The cell c is split into four new cells, each of these cells being formed by the new vertex and a facet of c.
Precondition: *tds.dimension()* $= 3$ and c is a cell of *tds*.

Vertex_handle *tds.insert_in_facet(Facet f)*

Creates a new vertex, inserts it in facet f and returns its handle. In dimension 3, the two incident cells are split into 3 new cells; in dimension 2, the facet is split into 3 facets.
Precondition: *tds.dimension()* ≥ 2 and f is a facet of *tds*.

Vertex_handle *tds.insert_in_facet(Cell_handle c, int i)*

Creates a new vertex, inserts it in facet i of c and returns its handle.
Precondition: *tds.dimension()* ≥ 2, $i \in \{0,1,2,3\}$ in dimension 3, $i = 3$ in dimension 2 and *(c,i)* is a facet of *tds*.

Vertex_handle *tds.insert_in_edge(Edge e)*

Creates a new vertex, inserts it in edge *e* and returns its handle. In dimension 3, all the incident cells are split into 2 new cells; in dimension 2, the 2 incident facets are split into 2 new facets; in dimension 1, the edge is split into 2 new edges.
Precondition: *tds.dimension()* ≥ 1 and *e* is an edge of *tds*.

Vertex_handle *tds.insert_in_edge(Cell_handle c, int i, int j)*

Creates a new vertex, inserts it in edge (i, j) of c and returns its handle.
Precondition: *tds.dimension()* ≥ 1. $i \neq j$, $i, j \in \{0,1,2,3\}$ in dimension 3, $i, j \in \{0,1,2\}$ in dimension 2, $i, j \in \{0,1\}$ in dimension 1 and *(c,i,j)* is an edge of *tds*.

Vertex_handle *tds.insert_increase_dimension(Vertex_handle star = Vertex_handle())*

Transforms a triangulation of the sphere S^d of $\mathbb{R}^{d+1}$ into the triangulation of the sphere S^{d+1} of $\mathbb{R}^{d+2}$ by adding a new vertex v: v is linked to all the vertices to triangulate one of the two half-spheres of dimension $(d+1)$. Vertex *star* is used to triangulate the second half-sphere (when there is an associated geometric triangulation, *star* is in fact the vertex associated with its infinite vertex). See Figure 40.8.
The numbering of the cells is such that, if f was a face of maximal dimension in the initial triangulation, then *(f,v)* (in this order) is the corresponding face in the new triangulation. This method can be used to insert the first two vertices in an empty triangulation.
A handle to v is returned.
Precondition: *tds.dimension()* $= d < 3$. When *tds.number_of_vertices()* > 0, $star \neq$ *Vertex_handle()* and *star* is a vertex of *tds*.

template <class CellIt>
Vertex_handle *tds.insert_in_hole(CellIt cell_begin, CellIt cell_end, Cell_handle begin, int i)*

Creates a new vertex by starring a hole. It takes an iterator range [*cell_begin*; *cell_end*[of *Cell_handles* which specifies a set of connected cells (resp. facets in dimension 2) describing a hole. (*begin*, *i*) is a facet (resp. an edge) on the boundary of the hole, that is, *begin* belongs to the set of cells (resp. facets) previously described, and *begin->neighbor(i)* does not. Then this function deletes all the cells (resp. facets) describing the hole, creates a new vertex v, and for each facet (resp. edge) on the boundary of the hole, creates a new cell (resp. facet) with v as vertex. v is returned.
Precondition: *tds.dimension()* ≥ 2, the set of cells (resp. facets) is connected, and its boundary is connected.

template <class CellIt>

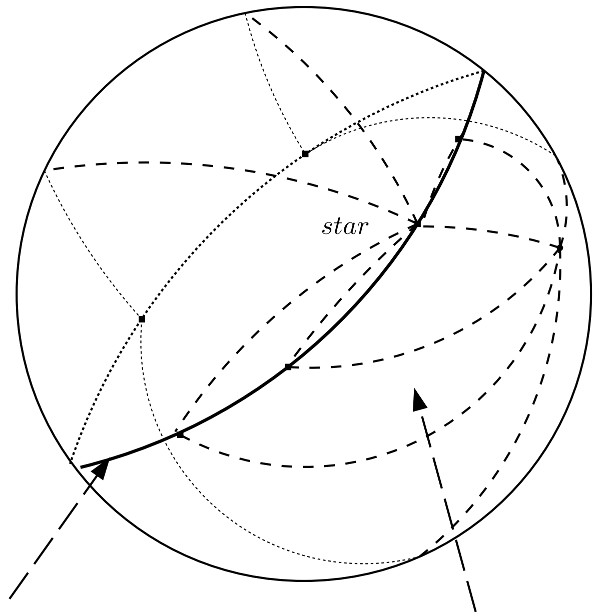


Figure 40.8: *insert_increase_dimension* (1-dimensional case).

Vertex_handle *tds.insert_in_hole(CellIt cell_begin,*
CellIt cell_end,
Cell_handle begin,
int i,
Vertex_handle newv)

Same as above, except that *newv* will be used as the new vertex, which must have been allocated previously with e.g. *create_vertex*.

Removal

void *tds.remove_decrease_dimension(Vertex_handle v, Vertex_handle w = v)*

This operation is the reciprocal of *insert_increase_dimension()*. It transforms a triangulation of the sphere S^d of $\mathbb{R}^{d+1}$ into the triangulation of the sphere S^{d-1} of $\mathbb{R}^d$ by removing the vertex v. Delete the cells incident to w, keep the others.
Precondition: *tds.dimension()* $= d \geq -1$. *tds.degree(v)* = *degree(w)* = *tds.number_of_vertices()* -1.

Cell_handle *tds.remove_from_maximal_dimension_simplex(Vertex_handle v)*

Removes *v*. The incident simplices of maximal dimension incident to *v* are replaced by a single simplex of the same dimension. This operation is exactly the reciprocal to *tds.insert_in_cell(v)* in dimension 3, *tds.insert_in_facet(v)* in dimension 2, and *tds.insert_in_edge(v)* in dimension 1.
Precondition: *tds.degree(v) = tds.dimension()+1*.

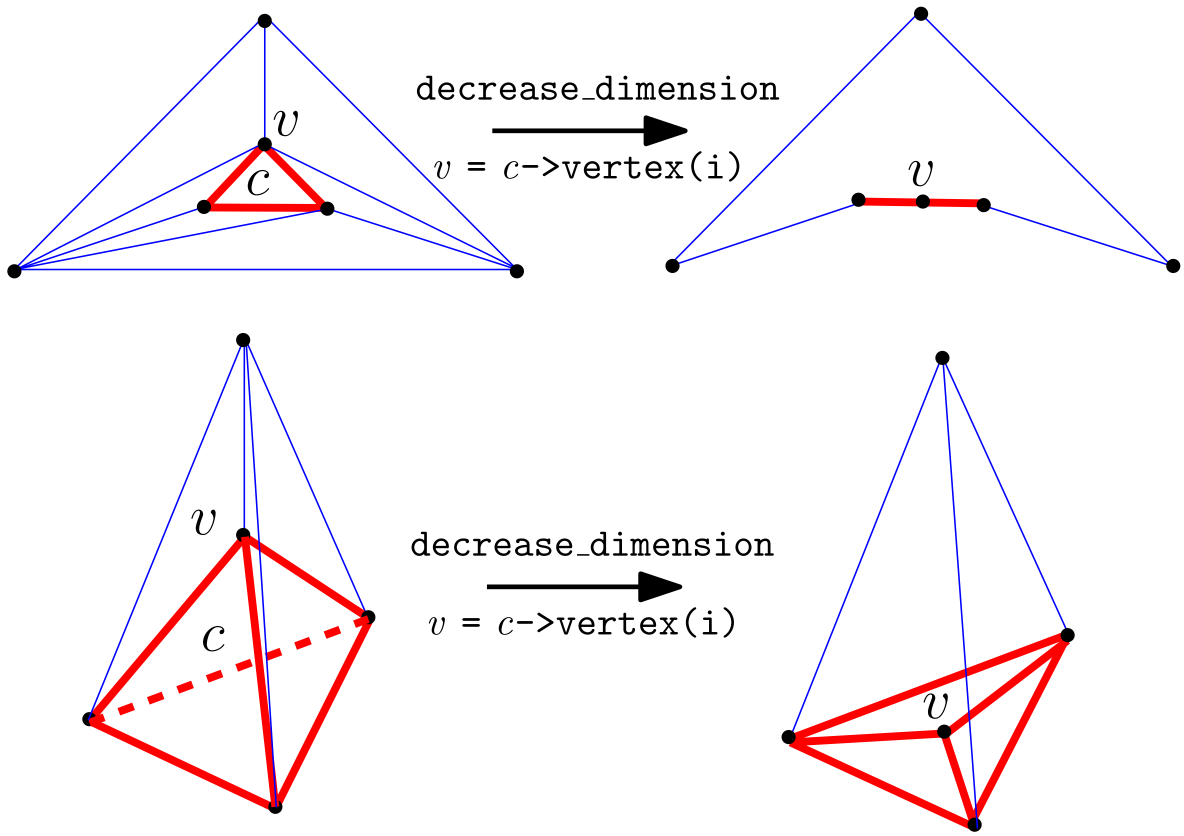


Figure 40.9: From an S^d data structure to an S^{d-1} data structure (top: $d == 2$, bottom: $d == 3$).

Dimension Manipulation

The following operation, `decrease_dimension`, is necessary when the displacement of a vertex decreases the dimension of the triangulation.

void *tds.decrease_dimension(Cell_handle c, int i)*

The link of a vertex v is formed by the facets disjoint from v that are included in the cells incident to v. When the link of v = *c->vertex(i)* contains all the other vertices, *decrease_dimension* crushes the triangulation of the sphere S^d of $\mathbb{R}^{d+1}$ onto the triangulation of the sphere S^{d-1} of $\mathbb{R}^d$ formed by the link of v augmented with the vertex v itself, for d==2,3; this one is placed on the facet *(c, i)* (see Fig. 40.9).
Precondition: The dimension must be 2 or 3. The degree of v must be equal to the total number of vertices of the triangulation data structure minus 1.

advanced

Other modifiers

The following modifiers can affect the validity of the triangulation data structure.

void	*tds.reorient()*	Changes the orientation of all cells of the triangulation data structure. *Precondition*: *tds.dimension()* ≥ 1.

Vertex_handle *tds.create_vertex(Vertex v = Vertex())*

Adds a copy of the vertex *v* to the triangulation data structure.

Vertex_handle *tds.create_vertex(Vertex_handle v)*

Creates a vertex which is a copy of the one pointed to by *v* and adds it to the triangulation data structure.

Cell_handle *tds.create_cell(Cell c = Cell())*

Adds a copy of the cell *c* to the triangulation data structure.

Cell_handle *tds.create_cell(Cell_handle c)*

Creates a cell which is a copy of the one pointed to by *c* and adds it to the triangulation data structure.

Cell_handle *tds.create_cell(Vertex_handle v0, Vertex_handle v1, Vertex_handle v2, Vertex_handle v3)*

Creates a cell and adds it into the triangulation data structure. Initializes the vertices of the cell, its neighbor handles being initialized with the default constructed handle.

Cell_handle *tds.create_cell(Vertex_handle v0,*
Vertex_handle v1,
Vertex_handle v2,
Vertex_handle v3,
Cell_handle n0,
Cell_handle n1,
Cell_handle n2,
Cell_handle n3)

Creates a cell, initializes its vertices and neighbors, and adds it into the triangulation data structure.

void *tds.delete_vertex(Vertex_handle v)*

Removes the vertex from the triangulation data structure.
Precondition: The vertex is a vertex of *tds*.

void *tds.delete_cell(Cell_handle c)*

Removes the cell from the triangulation data structure.
Precondition: The cell is a cell of *tds*.

template <class VertexIt>
void *tds.delete_vertices(VertexIt first, VertexIt last)*

Calls *delete_vertex* over an iterator range of value type *Vertex_handle*.

template <class CellIt>
void *tds.delete_cells(CellIt first, CellIt last)*

Calls *delete_cell* over an iterator range of value type *Cell_handle*.

advanced

Traversing the triangulation

Cell_iterator *tds.cells_begin() const*

Returns *cells_end()* when *tds.dimension()* < 3.

Cell_iterator *tds.cells_end() const*
Cell_iterator *tds.raw_cells_begin() const*

Low-level access to the cells, does not return *cells_end()* when *tds.dimension()* < 3.

Cell_iterator *tds.raw_cells_end() const*
Facet_iterator *tds.facets_begin() const*

Returns *facets_end()* when *tds.dimension()* < 2.

Facet_iterator *tds.facets_end() const*
Edge_iterator *tds.edges_begin() const*

Returns *edges_end()* when *tds.dimension()* < 1.

Edge_iterator *tds.edges_end() const*
Vertex_iterator *tds.vertices_begin() const*
Vertex_iterator *tds.vertices_end() const*

Cell and facet circulators

Cell_circulator *tds.incident_cells(Edge e) const* Starts at an arbitrary cell incident to *e*.
Precondition: *tds.dimension()* $= 3$

Cell_circulator *tds.incident_cells(Cell_handle c, int i, int j) const*

As above for edge *(i,j)* of *c*.

Cell_circulator *tds.incident_cells(Edge e, Cell_handle start) const*

Starts at cell *start*.
Precondition: *tds.dimension()* $= 3$ and *start* is incident to *e*.

Cell_circulator *tds.incident_cells(Cell_handle c, int i, int j, Cell_handle start) const*

As above for edge *(i,j)* of *c*.

The following circulators on facets are defined only in dimension 3, though facets are defined also in dimension 2: there are only two facets sharing an edge in dimension 2.

Facet_circulator *tds.incident_facets(Edge e) const*

Starts at an arbitrary facet incident to *e*.
Precondition: *tds.dimension()* $= 3$

Facet_circulator *tds.incident_facets(Cell_handle c, int i, int j) const*

As above for edge *(i,j)* of *c*.

Facet_circulator *tds.incident_facets(Edge e, Facet start) const*

Starts at facet *start*.
Precondition: *start* is incident to *e*.

Facet_circulator *tds.incident_facets(Edge e, Cell_handle start, int f) const*

Starts at facet of index *f* in *start*.

Facet_circulator *tds.incident_facets(Cell_handle c, int i, int j, Facet start) const*

As above for edge *(i,j)* of *c*.

Facet_circulator *tds.incident_facets(Cell_handle c, int i, int j, Cell_handle start, int f) const*

As above for edge *(i,j)* of *c* and facet *(start,f)*.

Traversal of the incident cells, facets and edges, and the adjacent vertices of a given vertex

template <class OutputIterator>
OutputIterator *tds.incident_cells(Vertex_handle v, OutputIterator cells) const*

Copies the *Cell_handle*s of all cells incident to *v* to the output iterator *cells*. Returns the resulting output iterator.
Precondition: *tds.dimension()* $= 3$, $v \neq$ *Vertex_handle()*, *tds.is_vertex(v)*.

template <class OutputIterator>
OutputIterator *tds.incident_facets(Vertex_handle v, OutputIterator facets) const*

Copies the *Facet*s incident to *v* to the output iterator *facets*. Returns the resulting output iterator.
Precondition: *tds.dimension()* > 1, $v \neq$ *Vertex_handle()*, *tds.is_vertex(v)*.

template <class OutputIterator>
OutputIterator *tds.incident_edges(Vertex_handle v, OutputIterator edges) const*

Copies all *Edge*s incident to *v* to the output iterator *edges*. Returns the resulting output iterator.
Precondition: *tds.dimension()* > 0, $v \neq$ *Vertex_handle()*, *tds.is_vertex(v)*.

template <class OutputIterator>

OutputIterator *tds.adjacent_vertices(Vertex_handle v, OutputIterator vertices) const*

Copies the *Vertex_handle*s of all vertices adjacent to v to the output iterator *vertices*. If *tds.dimension()* < 0, then do nothing. Returns the resulting output iterator.
Precondition: $v \neq$ *Vertex_handle()*, *tds.is_vertex(v)*.

size_type *tds.degree(Vertex_handle v) const*

Returns the degree of a vertex, that is, the number of incident vertices.
Precondition: $v \neq$ *Vertex_handle()*, *tds.is_vertex(v)*.

Traversal between adjacent cells

int *tds.mirror_index(Cell_handle c, int i) const*

Returns the index of c in its i^{th} neighbor.
Precondition: $i \in \{0,1,2,3\}$.

Vertex_handle *tds.mirror_vertex(Cell_handle c, int i) const*

Returns the vertex of the i^{th} neighbor of c that is opposite to c.
Precondition: $i \in \{0,1,2,3\}$.

Facet *tds.mirror_facet(Facet f) const* Returns the same facet seen from the other adjacent cell.

debugging support

Checking

bool *tds.is_valid(bool verbose = false) const*

Checks the combinatorial validity of the triangulation by checking the local validity of all its cells and vertices (see functions below). (See Section 40.1.) Moreover, the Euler relation is tested.
When *verbose* is set to *true*, messages are printed to give a precise indication on the kind of invalidity encountered.

bool *tds.is_valid(Vertex_handle v, bool verbose = false) const*

Checks the local validity of the adjacency relations of the triangulation. It also calls the *is_valid* member function of the vertex. When *verbose* is set to *true*, messages are printed to give a precise indication on the kind of invalidity encountered.

bool *tds.is_valid(Cell_handle c, bool verbose = false) const*

Checks the local validity of the adjacency relations of the triangulation. It also calls the *is_valid* member function of the cell. When *verbose* is set to *true*, messages are printed to give a precise indication on the kind of invalidity encountered.

debugging support

I/O

istream& *istream& is >> & tds* Reads a combinatorial triangulation from *is* and assigns it to *tds*

ostream& *ostream& os << tds* Writes *tds* into the stream *os*

The information stored in the *iostream* is: the dimension, the number of vertices, the number of cells, the indices of the vertices of each cell, then the indices of the neighbors of each cell, where the index corresponds to the preceding list of cells. When dimension < 3, the same information is stored for faces of maximal dimension instead of cells.

Has Models

CGAL::Triangulation_data_structure_3

See Also

TriangulationDataStructure_3::Vertex
TriangulationDataStructure_3::Cell

Concept

TriangulationDataStructure_3::Cell

Definition

The concept Cell stores four *Vertex_handle*s to its four vertices and four *Cell_handle*s to its four neighbors. The vertices are indexed 0, 1, 2, and 3 in consistent order. The neighbor indexed i lies opposite to vertex i.

In degenerate dimensions, cells are used to store faces of maximal dimension: in dimension 2, each cell represents only one facet of index 3, and 3 edges $(0,1)$, $(1,2)$ and $(2,0)$; in dimension 1, each cell represents one edge $(0,1)$. (See also Section 40.1.)

Types

The class Cell defines the following types.

typedef TriangulationDataStructure_3	*Triangulation_data_structure;*
typedef TriangulationDataStructure_3::Vertex_handle	*Vertex_handle;*
typedef TriangulationDataStructure_3::Cell_handle	*Cell_handle;*

Creation

In order to obtain new cells or destruct unused cells, the user must call the *create_cell()* and *delete_cell()* methods of the triangulation data structure.

Operations

Access Functions

Vertex_handle	*c.vertex(int i) const*	Returns the vertex i of c. *Precondition*: $i \in \{0,1,2,3\}$.
int	*c.index(Vertex_handle v) const*	Returns the index of vertex v in c. *Precondition*: v is a vertex of c.
bool	*c.has_vertex(Vertex_handle v) const*	Returns *true* if v is a vertex of c.
bool	*c.has_vertex(Vertex_handle v, int & i) const*	Returns *true* if v is a vertex of c, and computes its index i in c.
Cell_handle	*c.neighbor(int i) const*	Returns the neighbor i of c. *Precondition*: $i \in \{0,1,2,3\}$.
int	*c.index(Cell_handle n) const*	Returns the index corresponding to neighboring cell n. *Precondition*: n is a neighbor of c.
bool	*c.has_neighbor(Cell_handle n) const*	Returns *true* if n is a neighbor of c.

bool	*c.has_neighbor(Cell_handle n, int & i) const*	
		Returns *true* if *n* is a neighbor of *c*, and computes its index *i* in *c*.

Setting

void	*c.set_vertex(int i, Vertex_handle v)*	Sets vertex *i* to *v*. *Precondition*: $i \in \{0,1,2,3\}$.
void	*c.set_vertices(Vertex_handle v0, Vertex_handle v1, Vertex_handle v2, Vertex_handle v3)*	
		Sets the vertex pointers.
void	*c.set_neighbor(int i, Cell_handle n)*	Sets neighbor *i* to *n*. *Precondition*: $i \in \{0,1,2,3\}$.
void	*c.set_neighbors(Cell_handle n0, Cell_handle n1, Cell_handle n2, Cell_handle n3)*	
		Sets the neighbors pointers.

debugging support

Checking

bool	*c.is_valid(bool verbose = false, int level = 0) const*	
	User defined local validity checking function.	

debugging support

See Also

TriangulationDataStructure_3::Vertex.

Concept

TriangulationDataStructure_3::Vertex

Definition

The concept Vertex represents the vertex class of a 3D-triangulation data structure. It must define the types and operations listed in this section. Some of these requirements are of geometric nature, they are *optional* when using the triangulation data structure class alone. They become compulsory when the triangulation data structure is used as a layer for the geometric triangulation class. (See Section 40.2.)

Types

Vertex:: Point	*Optional for the triangulation data structure alone.*

The class Vertex defines types that are the same as some of the types defined by the triangulation data structure class *TriangulationDataStructure_3*.

typedef TriangulationDataStructure_3	*Triangulation_data_structure;*
typedef TriangulationDataStructure_3::Vertex_handle	*Vertex_handle;*
typedef TriangulationDataStructure_3::Cell_handle	*Cell_handle;*

Creation

In order to obtain new vertices or destruct unused vertices, the user must call the *create_vertex()* and *delete_vertex()* methods of the triangulation data structure.

Operations

Access Functions

Cell_handle	*v.cell() const*	Returns a cell of the triangulation having *v* as vertex.
Point	*v.point() const*	Returns the point stored in the vertex. *Optional for the triangulation data structure alone.*

Setting

void	*v.set_cell(Cell_handle c)*	Sets the incident cell to *c*.
void	*v.set_point(Point p)*	Sets the point to *p*. *Optional for the triangulation data structure alone.*

debugging support

Checking

bool *v.is_valid(bool verbose = false) const*

Checks the validity of the vertex. Must check that its incident cell has this vertex. The validity of the base vertex is also checked.
When *verbose* is set to *true*, messages are printed to give a precise indication on the kind of invalidity encountered.

debugging support

See Also

TriangulationDataStructure_3::Cell.

TriangulationDSCellBase_3

Definition

At the base level (see Sections 39.4 and 40.2), a cell stores handles to its four vertices and to its four neighbor cells. The vertices and neighbors are indexed 0, 1, 2 and 3. Neighbor *i* lies opposite to vertex *i*.

Since the Triangulation data structure is the class which defines the handle types, the cell base class has to be somehow parameterized by the Triangulation data structure. But since it is itself parameterized by the cell and vertex base classes, there is a cycle in the definition of these classes. In order to break the cycle, the base classes for vertex and cell which are given as arguments for the Triangulation data structure use *void* as Triangulation data structure parameter, and the Triangulation data structure then uses a *rebind*-like mechanism (similar to the one specified in *std::allocator*) in order to put itself as parameter to the vertex and cell classes. The *rebound* base classes so obtained are the classes which are used as base classes for the final vertex and cell classes. More information can be found in Section 40.2.

Types

The concept TriangulationDSCellBase_3 has to provide the following types.

TriangulationDSCellBase_3:: template <typename TDS2> struct Rebind_TDS

This nested template class has to define a type *Other* which is the *rebound* cell, that is, the one whose *Triangulation_data_structure* will be the actually used one. The *Other* type will be the real base class of *Triangulation_data_structure_3::Cell*.

typedef TriangulationDataStructure_3	*Triangulation_data_structure;*
typedef TriangulationDataStructure_3::Vertex_handle	*Vertex_handle;*
typedef TriangulationDataStructure_3::Cell_handle	*Cell_handle;*
typedef TriangulationDataStructure_3::Cell_data	*TDS_data;*

Creation

TriangulationDSCellBase_3 c;

Default constructor

TriangulationDSCellBase_3 c(Vertex_handle v0, Vertex_handle v1, Vertex_handle v2, Vertex_handle v3);

Initializes the vertices with *v0, v1, v2, v3*. Neighbors are initialized to the default constructed handle.

TriangulationDSCellBase_3 c(Vertex_handle v0,
Vertex_handle v1,
Vertex_handle v2,
Vertex_handle v3,
Cell_handle n0,
Cell_handle n1,
Cell_handle n2,

Cell_handle n3)

Initializes the vertices with *v0, v1, v2, v3* and the neighbors with *n0, n1, n2, n3*.

Access Functions

Vertex_handle	*c.vertex(int i) const*	Returns the vertex i of c. *Precondition*: $i \in \{0,1,2,3\}$.
int	*c.index(Vertex_handle v) const*	Returns the index of v. *Precondition*: v is a vertex of c
bool	*c.has_vertex(Vertex_handle v)*	True iff v is a vertex of c.
bool	*c.has_vertex(Vertex_handle v, int & i) const*	Returns *true* if v is a vertex of c, and computes its index i in c.
Cell_handle	*c.neighbor(int i) const*	Returns the neighbor i of c. *Precondition*: $i \in \{0,1,2,3\}$.
int	*c.index(Cell_handle n) const*	Returns the index of cell n in c. *Precondition*: n is a neighbor of c.
bool	*c.has_neighbor(Cell_handle n)*	Returns *true* if n is a neighbor of c.
bool	*c.has_neighbor(Cell_handle n, int & i) const*	Returns *true* if n is a neighbor of c, and computes its index i in c.

Setting

void	*c.set_vertex(int i, Vertex_handle v)*	Sets vertex i to v. *Precondition*: $i \in \{0,1,2,3\}$.
void	*c.set_vertices()*	Sets the vertices to the default constructed handle.
void	*c.set_vertices(Vertex_handle v0, Vertex_handle v1, Vertex_handle v2, Vertex_handle v3)*	Sets the vertices.
void	*c.set_neighbor(int i, Cell_handle n)*	Sets neighbor i to n. *Precondition*: $i \in \{0,1,2,3\}$.
void	*c.set_neighbors()*	Sets the neighbors to the default constructed handle.
void	*c.set_neighbors(Cell_handle n0, Cell_handle n1, Cell_handle n2, Cell_handle n3)*	Sets the neighbors.

debugging support

Checking

bool *c.is_valid(bool verbose = false, int level = 0) const*

Performs any desired geometric test on a cell. When *verbose* is set to *true*, messages are printed to give a precise indication of the kind of invalidity encountered. *level* increases the level of testing.

debugging support

advanced

Various

*void** *c.for_compact_container() const*
void&* *c.for_compact_container()*

These member functions are required by *Triangulation_data_structure_3* because it uses *Compact_container* to store its cells. See the documentation of *Compact_container* for the exact requirements.

TDS_data& *c.tds_data()*
TDS_data *c.tds_data() const*

These functions are used internally by the triangulation data structure. The user is not encouraged to use them directly as they may change in the future.

advanced

I/O

istream&	*istream& is >> & c*	Inputs the possible non combinatorial information given by the cell.
ostream&	*ostream& os << c*	Outputs the possible non combinatorial information given by the cell.

Has Models

CGAL::Triangulation_ds_cell_base_3
CGAL::Triangulation_cell_base_3
CGAL::Triangulation_cell_base_with_info_3

See Also

TriangulationDSVertexBase_3
TriangulationVertexBase_3
TriangulationHierarchyVertexBase_3
TriangulationCellBase_3

TriangulationDSVertexBase_3

Definition

At the bottom level of 3D-triangulations (see Sections 39.4 and 40.2), a vertex provides access to one of its incident cells through a handle.

Note that when you use the triangulation data structure as parameter of a geometric triangulation, the vertex base class has additional geometric requirements : it has to match the *TriangulationVertexBase_3* concept.

Since the Triangulation data structure is the class which defines the handle types, the vertex base class has to be somehow parameterized by the Triangulation data structure. But since it is itself parameterized by the cell and vertex base classes, there is a cycle in the definition of these classes. In order to break the cycle, the base classes for vertex and cell which are given as arguments for the Triangulation data structure use *void* as Triangulation data structure parameter, and the Triangulation data structure then uses a *rebind*-like mechanism (similar to the one specified in *std::allocator*) in order to put itself as parameter to the vertex and cell classes. The *rebound* base classes so obtained are the classes which are used as base classes for the final vertex and cell classes. More information can be found in Section 40.2.

Types

The class TriangulationDSVertexBase_3 has to define the following types.

TriangulationDSVertexBase_3:: template <typename TDS2> struct Rebind_TDS

This nested template class has to define a type *Other* which is the *rebound* vertex, that is, the one whose *Triangulation_data_structure* will be the actually used one. The *Other* type will be the real base class of *Triangulation_data_structure_3::Vertex*.

typedef TriangulationDataStructure_3	*Triangulation_data_structure;*
typedef TriangulationDataStructure_3::Vertex_handle	*Vertex_handle;*
typedef TriangulationDataStructure_3::Cell_handle	*Cell_handle;*

Creation

TriangulationDSVertexBase_3 v;	Default constructor.
TriangulationDSVertexBase_3 v(Cell_handle c);	Constructs a vertex pointing to cell *c*.

Operations

Access Functions

Cell_handle	*v.cell() const*	Returns the pointer to an incident cell

Setting

void	*v.set_cell(Cell_handle c)*	Sets the incident cell.

debugging support

Checking

bool *v.is_valid(bool verbose=false, int level=0) const*

Performs any desired test on a vertex. Checks that the pointer to an incident cell is not the default constructed handle.

debugging support

Various

*void** *v.for_compact_container() const*
void&* *v.for_compact_container()*

These member functions are required by *Triangulation_data_structure_3* because it uses *Compact_container* to store its cells. See the documentation of *Compact_container* for the exact requirements.

I/O

istream&	*istream& is >> & v*	Inputs the non-combinatorial information given by the vertex.
ostream&	*ostream& os << v*	Outputs the non-combinatorial information given by the vertex.

Has Models

CGAL::Triangulation_ds_vertex_base_3
CGAL::Triangulation_vertex_base_3
CGAL::Triangulation_vertex_base_with_info_3
CGAL::Triangulation_hierarchy_vertex_base_3

See Also

TriangulationVertexBase_3
TriangulationHierarchyVertexBase_3
TriangulationDSCellBase_3
TriangulationCellBase_3

CGAL::Triangulation_data_structure_3<TriangulationDSVertexBase_3,TriangulationDSCellBase_3>

Class

Definition

The class *Triangulation_data_structure_3* stores a 3D-triangulation data structure and provides the optional geometric functionalities to be used as a parameter for a 3D-geometric triangulation (see Chapter 39).

The vertices and cells are stored in two nested containers, which are implemented using *CGAL::Compact_container*. The class may offer some flexibility for the choice of container in the future, in the form of additional template parameters.

#include <CGAL/Triangulation_data_structure_3.h>

Parameters

It is parameterized by base classes for vertices and cells which have to match the requirements for the concepts *TriangulationDSCellBase_3* and *TriangulationDSVertexBase_3* respectively (see page 2734 and page 2737).

They have the default values *Triangulation_ds_vertex_base_3<>* and *Triangulation_ds_cell_base_3<>* respectively.

Is Model for the Concepts

TriangulationDataStructure_3

Inherits From

CGAL::Triangulation_utils_3

The class *Triangulation_utils_3* defines basic computations on indices of vertices and neighbors of cells.

advanced

In addition to the interface documented in the concept, the class offers the following types and functions.

Types

typedef CGAL::Compact_container<Vertex>

Vertex_range; Vertex container type.

typedef CGAL::Compact_container<Cell>

Cell_range; Cell container type.

Operations

Cell_range&	*tds.cells() const*	Returns a reference to the container of cells.
Cell_range&	*tds.cells()*	Returns a reference to the container of cells.
Vertex_range&	*tds.vertices() const*	Returns a reference to the container of vertices.
Vertex_range&	*tds.vertices()*	Returns a reference to the container of vertices.

advanced

See Also

CGAL::Triangulation_ds_vertex_base_3
CGAL::Triangulation_ds_cell_base_3
CGAL::Triangulation_vertex_base_with_info_3
CGAL::Triangulation_cell_base_with_info_3

CGAL::Triangulation_ds_cell_base_3<>

Class

Definition

The class *Triangulation_ds_cell_base_3* is a model for the concept *TriangulationDSCellBase_3* to be used by *Triangulation_data_structure_3*.

#include <CGAL/Triangulation_ds_cell_base_3.h>

Is Model for the Concepts

TriangulationDSCellBase_3

See Also

CGAL::Triangulation_cell_base_3
CGAL::Triangulation_ds_vertex_base_3
CGAL::Triangulation_cell_base_with_info_3

Class

CGAL::Triangulation_ds_vertex_base_3<>

Definition

The class *Triangulation_ds_vertex_base_3* can be used as the base vertex for a 3D-triangulation data structure, it is a model of the concept *TriangulationDSVertexBase_3*.

Note that if the triangulation data structure is used as a parameter of a geometric triangulation (Section 40.2 and Chapter 39), then the vertex base class has to fulfill additional geometric requirements, i.e. it has to be a model of the concept *TriangulationVertexBase_3*.

This base class can be used directly or can serve as a base to derive other base classes with some additional attributes (a color for example) tuned for a specific application.

#include <CGAL/Triangulation_ds_vertex_base_3.h>

Is Model for the Concepts

TriangulationDSVertexBase_3

See Also

CGAL::Triangulation_vertex_base_3
CGAL::Triangulation_ds_cell_base_3
CGAL::Triangulation_vertex_base_with_info_3

CGAL::Triangulation_utils_3

Definition

The class *Triangulation_utils_3* defines operations on the indices of vertices and neighbors within a cell.

#include <CGAL/Triangulation_utils_3.h>

Operations

unsigned int *next_around_edge(unsigned int i, unsigned int j) const*

In dimension 3, index of the neighbor *n* that is next to the current cell, when turning positively around an oriented edge whose endpoints are indexed *i* and *j*. According to the usual numbering of vertices and neighbors in a given cell, it is also the index of the vertex opposite to this neighbor *n*. (see Figure 40.10).
Precondition: *(i < 4)* `&&` *(j < 4)* `&&` *(i != j)*.

unsigned int *ccw(unsigned int i) const*

Has a meaning only in dimension 2.
Computes the index of the vertex that is next to the vertex numbered *i* in counterclockwise direction. (see Figure 40.10).
Precondition: *i<3*.

unsigned int *cw(unsigned int i) const*

Same for clockwise.

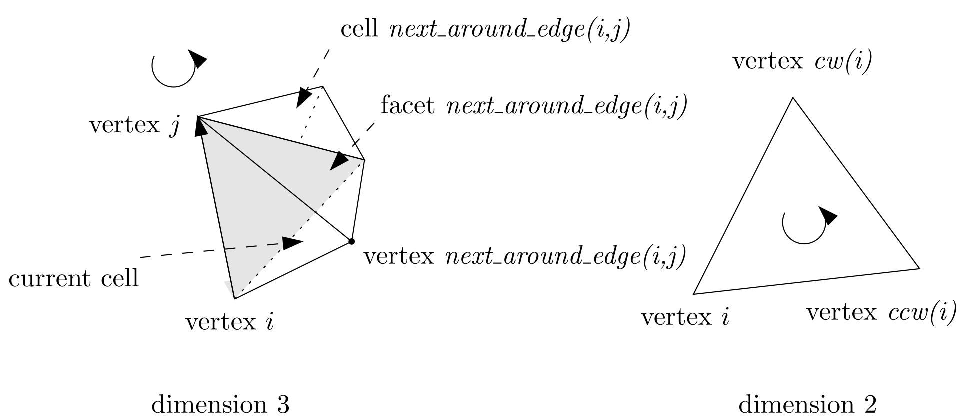


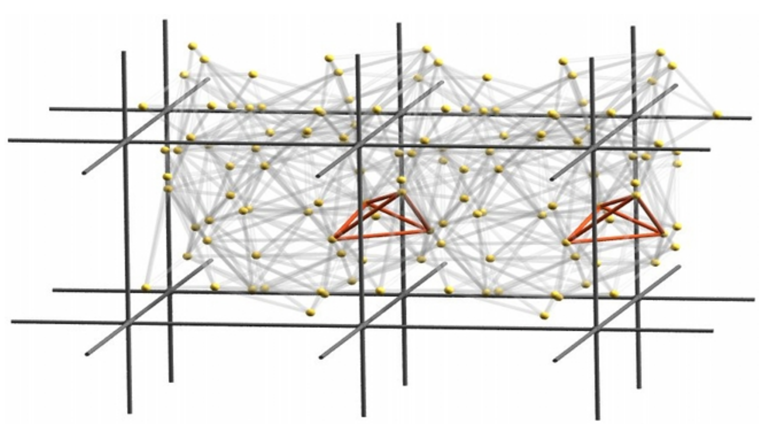
Figure 40.10: Operations on indices.

Chapter 41

3D Periodic Triangulations

Manuel Caroli and Monique Teillaud

Contents



The periodic 3D-triangulation class of CGAL is designed to represent the triangulations of a set of points in the three-dimensional flat torus. The triangulation forms a partition of the space it is computed in. It is a simplicial complex, i.e. it contains all incident j-simplices ($j < k$) of any k-simplex and two k-simplices either do not intersect or share a common j-face, $j < k$. The occurring simplices of dimension up to three are called *vertex*, *edge*, *facet*, and *cell*, respectively.

41.1 The Flat Torus

The 3D Periodic Triangulation package computes triangulations in the space $\mathbb{T}_c^3$, which is defined as follows: Let $c \in \mathbb{R} \setminus \{0\}$ and G be the group $(c \cdot \mathbb{Z}^3, +)$, where $c \cdot \mathbb{Z}$ denotes the set containing all integer multiples of c. The flat torus is the quotient space: $\mathbb{T}_c^3 := \mathbb{R}^3/G$. The parameter c defines the period.

The elements of $\mathbb{T}_c^3$ are the equivalence classes of sets of points in $\mathbb{R}^3$. We call these points *representatives* of an element of $\mathbb{T}_c^3$. The implementation works not directly on elements of $\mathbb{T}_c^3$ but on some representatives in $\mathbb{R}^3$. So there need to be distinguished representatives to work on. Given α, β, and γ, the cube $[\alpha, \alpha + c) \times [\beta, \beta + c) \times [\gamma, \gamma + c)$ contains exactly one representative of each element in $\mathbb{T}_c^3$. We call it *original domain*. From now on, when we talk about *points*, we generally mean representatives of elements of $\mathbb{T}_c^3$ that lie inside the original domain. Note that any input point is required to be an element of the half-open cube representing the original domain as defined above.

There are simplices containing points inside the original domain but also points outside it. The points outside the original domain are periodic copies of points inside the original domain. So, to specify a simplex we need points together with some additional information that determines the respective periodic copy of each point. The set of representatives of an element of $\mathbb{T}_c^3$ is a cubic point grid. We address each representative by a three-dimensional integer vector (o_x, o_y, o_z), called *offset*. It represents the number of periods a representative in the original domain must be translated in x-, y-, and z-direction. The vector $(0,0,0)$ corresponds to the representative in the original domain. To specify a k-simplex we need $k+1$ point-offset pairs (cf. Fig. 41.1).

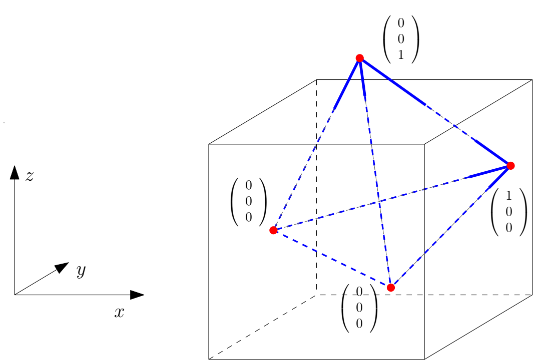


Figure 41.1: Offsets in a cell.

41.2 Representation

A triangulation is a collection of vertices and cells that are linked together through incidence and adjacency relations. Each cell gives access to its four incident vertices, their corresponding offsets, and to its four adjacent cells. Each vertex gives access to one of its incident cells.

The four vertices of a cell are indexed with 0, 1, 2 and 3 in positive orientation. The orientation of a simplex in

$\mathbb{T}_c^3$ is defined as the orientation of the corresponding simplex in $\mathbb{R}^3$ given by representatives determined by the respective offsets (see Figure 41.2).

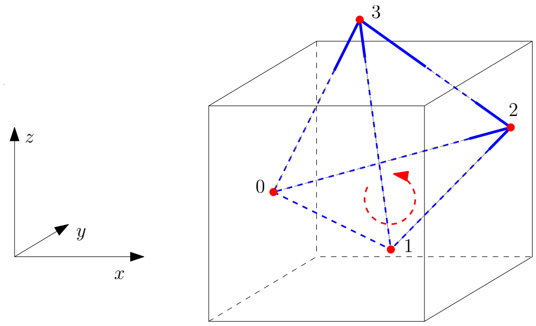


Figure 41.2: Orientation of a cell.

As in the underlying combinatorial triangulation (see Chapter 40), the neighbors of a cell are indexed with 0, 1, 2, 3 in such a way that the neighbor indexed by i is opposite to the vertex with the same index. Also edges (1-faces) and facets (2-faces) are not explicitly represented: a facet is given by a cell and an index (the facet i of a cell c is the facet of c that is opposite to the vertex with index i) and an edge is given by a cell and two indices (the edge *(i,j)* of a cell c is the edge whose endpoints are the vertices of c with indices i and j). See Figure 40.1.

Some point sets do not admit a triangulation in $\mathbb{T}_c^3$. In this case we use 27 periodic copies of the point set arranged in a cube of edge length $3c$. Any point set constructed in this way has a triangulation in $\mathbb{R}^3/G'$ with $G' = ((3c \cdot \mathbb{Z})^3, +)$ [CT09]. So we compute the triangulation in this space, which is a *27-sheeted covering space* of $\mathbb{T}_c^3$ (see Figure 41.3).

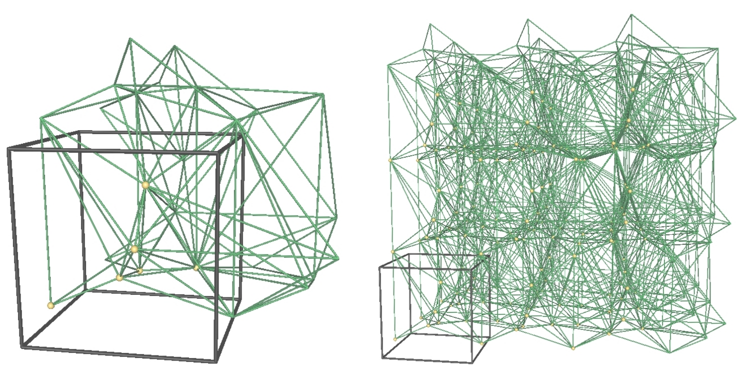


Figure 41.3: The same periodic triangulation in the 1-sheeted covering space and the 27-sheeted covering space.

The machinery that manages the copies is largely hidden from the user. However there are some effects that cannot be ignored. For example if the point set does not permit a triangulation in $\mathbb{T}_c^3$ then the combinatorial iterators (*Cell_iterator*, *Facet_iterator*, *Edge_iterator*, and *Vertex_iterator*) return all simplices that are internally stored, which correspond to 27 periodic copies of each geometric primitive (Tetrahedron, Triangle, Segment, and Point). This is necessary to ensure consistency in the adjacency relations. In case it is desired to have only one periodic copy of each primitive, we provide *geometric* iterators. They return geometric primitives of the triangulation without relations between them. Another effect is that when the algorithm switches from the 27-sheeted covering space to the 1-sheeted covering space, the *Vertex_handle*s and *Cell_handle*s referencing deleted items become invalid.

In the data structure each vertex stores the input point it corresponds to. If we are computing in the 27-sheeted covering space, each vertex stores the representative *inside* the original domain it corresponds to. So, the 27 vertices corresponding to the same element of $\mathbb{T}_c^3$ all store the same representative in $\mathbb{R}^3$, and not different periodic copies.

Validity A periodic triangulation is said to be *locally valid* iff

(a)-(b) Its underlying combinatorial graph, the triangulation data structure, is *locally valid* (see Section 40.1 of Chapter 40)
(c) Any cell has its vertices ordered according to positive orientation. See Figure 41.2.

41.3 Delaunay Triangulation

The class *Periodic_3_Delaunay_triangulation_3* implements Delaunay triangulations of point sets in $\mathbb{T}_c^3$.

Delaunay triangulations have the *empty sphere property*, that is, the circumscribing sphere of each cell does not contain any other vertex of the triangulation in its interior. These triangulations are uniquely defined except in degenerate cases where five points are co-spherical. Note however that the CGAL implementation computes a unique triangulation even in these cases [DT03].

This implementation is fully dynamic: it supports both insertions of points and vertex removal.

41.4 Triangulation Hierarchy

The class *Periodic_3_triangulation_hierarchy_3* is the adaptation of the hierarchical structure described in chapter 39 to the periodic case.

41.5 Software Design

We have chosen the prefix "Periodic_3" to emphasize that the triangulation is periodic in all three directions of space. There are also "cylindrical" periodicities where the triangulation is periodic only in one or two directions of space.

The two main classes *Periodic_3_Delaunay_triangulation_3* and *Periodic_3_triangulation_3* provide high-level geometric functionality and are responsible for the geometric validity. *Periodic_3_Delaunay_triangulation_3* contains all the functionality that is special to Delaunay triangulations, such as point insertion and vertex removal, the side-of-sphere test, finding the conflicting region of a given point, dual functions etc. *Periodic_3_triangulation_3* contains all the functionality that is common to triangulations in general, such as location of a point in the triangulation [DPT02], access functions, geometric queries like the orientation test etc.

They are built as layers on top of a triangulation data structure, which stores their combinatorial structure. This separation between the geometry and the combinatorics is reflected in the software design by the fact that the triangulation classes take two template parameters:

- the **geometric traits** class, which provides the type of points to use as well as the elementary operations on them (predicates and constructions). Furthermore it contains the offset type. The concept for this parameter is described in more detail in Section 41.5.1 and as the concept *Periodic_3DelaunayTriangulationTraits_3* in the reference manual on page 2783.

- the **triangulation data structure** class, which stores the combinatorial structure, described in Section 41.5.2 and in more detail in Chapter 40. The triangulation data structure needs models of the concepts *Periodic_3TriangulationDSCellBase_3* and *Periodic_3TriangulationDSVertexBase_3* as template parameters.

41.5.1 The Geometric Traits Parameter

The first template parameter of the Delaunay triangulation class *Periodic_3_Delaunay_triangulation_3< Periodic_3DelaunayTriangulationTraits_3, TriangulationDataStructure_3>* is the geometric traits class, described by the concept *Periodic_3DelaunayTriangulationTraits_3*. It is different to the DelaunayTriangulationTraits_3 (see chapter 39.4.1) in that it implements all objects, predicates and constructions with using offsets.

The class *Periodic_3_triangulation_traits_3<Traits,Periodic_3Offset_3>* provides the required functionality. It expects two template parameters: A model of the concept *DelaunayTriangulationTraits_3* and a model of the concept *Periodic_3Offset_3*.

The kernels *Cartesian*, *Homogeneous*, *Simple_cartesian*, *Simple_homogeneous* and *Filtered_kernel* can all be used as models for *Traits*. *Periodic_3_triangulation_traits_3* provides exact predicates and exact constructions if *Traits* does. It provides exact predicates but not exact constructions if *Filtered_kernel<CK>* with *CK* an inexact kernel is used as its first template parameter. Using *Exact_predicates_inexact_constructions_kernel* (page **??**) as *Traits* provides fast and exact predicates and not exact constructions, using *Exact_predicates_exact_constructions_kernel* (page **??**) provides fast and exact predicates and exact constructions. The latter is recommended if the dual constructions and constructions of points, segments, triangles, and tetrahedra are used.

The second parameter *Periodic_3Offset_3* defaults to *Periodic_3_offset_3*.

41.5.2 The Triangulation Data Structure Parameter

The second template parameter of the main classes *Periodic_3_triangulation_3* and *Periodic_3_Delaunay_triangulation_3* is a triangulation data structure class. This class can be seen as a container for the cells and vertices maintaining incidence and adjacency relations (see Chapter 40). A model of this triangulation data structure is *Triangulation_data_structure_3*, and it is described by the *TriangulationDataStructure_3* concept. This model is itself parameterized by a vertex base class and a cell base class, which gives the possibility to customize the vertices and cells used by the triangulation data structure, and hence by the geometric triangulation using it. To represent periodic triangulations the cell base and vertex base classes need to meet the concepts *Periodic_3TriangulationDSCellBase_3* and *Periodic_3TriangulationDSVertexBase_3*.

A default value for the triangulation data structure parameter is provided in all the triangulation classes, so it does not need to be specified by the user unless he wants to use a different triangulation data structure or a different vertex or cell base class.

41.5.3 Flexibility of the Design

Periodic_3_triangulation_3 uses the *TriangulationDataStructure_3* in essentially the same way as *Triangulation_3*. That is why the flexibility described in 39.4 is applicable in exactly the same way. Also the classes *Triangulation_vertex_base_with_info_3* and *Triangulation_cell_base_with_info_3* can be reused directly, see also Example 41.6.2.

41.6 Examples

41.6.1 Basic Example

This example shows the incremental construction of a 3D Delaunay triangulation, the location of a point and how to perform elementary operations on indices in a cell. It uses the default parameter of the *Periodic_3_Delaunay_triangulation_3* class for the triangulation data structure.

```
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/Periodic_3_triangulation_traits_3.h>
#include <CGAL/Periodic_3_Delaunay_triangulation_3.h>

#include <iostream>
#include <fstream>
#include <cassert>
#include <list>
#include <vector>

typedef CGAL::Exact_predicates_inexact_constructions_kernel K;
typedef CGAL::Periodic_3_triangulation_traits_3<K> GT;

typedef CGAL::Periodic_3_Delaunay_triangulation_3<GT> PDT;

typedef PDT::Cell_handle       Cell_handle;
typedef PDT::Vertex_handle     Vertex_handle;
typedef PDT::Locate_type       Locate_type;
typedef PDT::Point             Point;
typedef PDT::Iso_cuboid        Iso_cuboid;

int main()
{
  Iso_cuboid domain(-1,-1,-1,2,2,2);  // The cube for the periodic domain

  // construction from a list of points :
  std::list<Point> L;
  L.push_front(Point(0,0,0));
  L.push_front(Point(1,0,0));
  L.push_front(Point(0,1,0));

  PDT T(L.begin(), L.end(), domain); // Put the domain with the
    constructor

  PDT::size_type n = T.number_of_vertices();

  // insertion from a vector :
  std::vector<Point> V(3);
  V[0] = Point(0,0,1);
  V[1] = Point(1,1,1);
  V[2] = Point(-1,-1,-1);

  n = n + T.insert(V.begin(), V.end());

  assert( n == 6 );       // 6 points have been inserted
```

```
  assert( T.is_valid() ); // checking validity of T

  Locate_type lt;
  int li, lj;
  Point p(0,0,0);
  Cell_handle c = T.locate(p, lt, li, lj);
  // p is the vertex of c of index li :
  assert( lt == PDT::VERTEX );
  assert( c->vertex(li)->point() == p );

  Vertex_handle v = c->vertex( (li+1)&3 );
  // v is another vertex of c
  Cell_handle nc = c->neighbor(li);
  // nc = neighbor of c opposite to the vertex associated with p
  // nc must have vertex v :
  int nli;
  assert( nc->has_vertex( v, nli ) );
  // nli is the index of v in nc

  std::ofstream oFileT("output.tri",std::ios::out);
  // writing file output;
  oFileT << T;

  PDT T1;
  std::ifstream iFileT("output.tri",std::ios::in);
  // reading file output;
  iFileT >> T1;
  assert( T1.is_valid() );
  assert( T1.number_of_vertices() == T.number_of_vertices() );
  assert( T1.number_of_cells() == T.number_of_cells() );

  return 0;
}
```

File: examples/Periodic_3_triangulation_3/simple_example.cpp

41.6.2 Changing the Vertex Base

The following two examples show how the user can plug his own vertex base in a triangulation. Changing the cell base is similar.

Adding a Color

If the user does not need to add a type in a vertex that depends on the *TriangulationDataStructure_3* (e.g. a *Vertex_handle* or *Cell_handle*), then he can use the *Triangulation_vertex_base_with_info_3* class to add his own information easily in the vertices. The example below shows how to add a *CGAL::Color* this way.

```
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/Periodic_3_triangulation_filtered_traits_3.h>
#include <CGAL/Periodic_3_Delaunay_triangulation_3.h>
```

```
#include <CGAL/Triangulation_vertex_base_with_info_3.h>
#include <CGAL/IO/Color.h>

typedef CGAL::Exact_predicates_inexact_constructions_kernel K;
typedef CGAL::Periodic_3_triangulation_filtered_traits_3<K> GT;

typedef CGAL::Periodic_3_triangulation_ds_vertex_base_3<> VbDS;
typedef CGAL::Triangulation_vertex_base_3<GT,VbDS> Vb;

typedef CGAL::Periodic_3_triangulation_ds_cell_base_3<> CbDS;
typedef CGAL::Triangulation_cell_base_3<GT,CbDS> Cb;

typedef CGAL::Triangulation_vertex_base_with_info_3<CGAL::Color, GT, Vb>
   VbInfo;
typedef CGAL::Triangulation_data_structure_3<VbInfo, Cb>    TDS;
typedef CGAL::Periodic_3_Delaunay_triangulation_3<GT, TDS>  PDT;

typedef PDT::Point   Point;

int main()
{
  PDT T;

  T.insert(Point(0,0,0));
  T.insert(Point(.1,0,0));
  T.insert(Point(0,.1,0));
  T.insert(Point(0,0,.1));
  T.insert(Point(.2,.2,.2));
  T.insert(Point(.9,0,.1));

  // Set the color of finite vertices of degree 6 to red.
  PDT::Vertex_iterator vit;
  for (vit = T.vertices_begin(); vit != T.vertices_end(); ++vit)
    if (T.degree(vit) == 6)
      vit->info() = CGAL::RED;

  return 0;
}
```

File: examples/Periodic_3_triangulation_3/colored_vertices.cpp

Adding Handles

If the user needs to add a type in a vertex that depends on the *TriangulationDataStructure_3* (e.g. a *Vertex_handle* or *Cell_handle*), then he has to derive his own vertex base class, as the following example shows.

```
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/Periodic_3_triangulation_filtered_traits_3.h>
#include <CGAL/Periodic_3_Delaunay_triangulation_3.h>
#include <CGAL/Periodic_3_triangulation_ds_vertex_base_3.h>
#include <CGAL/Triangulation_vertex_base_3.h>
```

```
template < class GT, class VbDS,
           class Vb = CGAL::Triangulation_vertex_base_3<GT,VbDS> >
class My_vertex_base
  : public Vb
{
public:
  typedef typename Vb::Vertex_handle  Vertex_handle;
  typedef typename Vb::Cell_handle    Cell_handle;
  typedef typename Vb::Point          Point;

  template < class TDS2 >
  struct Rebind_TDS {
    typedef typename Vb::template Rebind_TDS<TDS2>::Other  Vb2;
    typedef My_vertex_base<GT, Vb2>                        Other;
  };

  My_vertex_base() {}

  My_vertex_base(const Point& p)
    : Vb(p) {}

  My_vertex_base(const Point& p, Cell_handle c)
    : Vb(p, c) {}

  Vertex_handle   vh;
  Cell_handle     ch;
};


typedef CGAL::Exact_predicates_inexact_constructions_kernel K;
typedef CGAL::Periodic_3_triangulation_filtered_traits_3<K> GT;

typedef CGAL::Periodic_3_triangulation_ds_vertex_base_3<> VbDS;
typedef CGAL::Periodic_3_triangulation_ds_cell_base_3<> CbDS;
typedef CGAL::Triangulation_cell_base_3<GT,CbDS> Cb;

typedef CGAL::Triangulation_data_structure_3<My_vertex_base<GT,VbDS>, Cb>
    TDS;
typedef CGAL::Periodic_3_Delaunay_triangulation_3<GT,TDS> PDT;

typedef PDT::Vertex_handle    Vertex_handle;
typedef PDT::Point            Point;

int main()
{
  PDT T;

  Vertex_handle v0 = T.insert(Point(0,0,0));
  Vertex_handle v1 = T.insert(Point(.1,0,0));
  Vertex_handle v2 = T.insert(Point(0,.1,0));
  Vertex_handle v3 = T.insert(Point(0,0,.1));
  Vertex_handle v4 = T.insert(Point(.2,.2,.2));
  Vertex_handle v5 = T.insert(Point(.9,0,.1));
```

```
  // Now we can link the vertices as we like.
  v0->vh = v1;
  v1->vh = v2;
  v2->vh = v3;
  v3->vh = v4;
  v4->vh = v5;
  v5->vh = v0;

  return 0;
}
```

File: examples/Periodic_3_triangulation_3/periodic_adding_handles.cpp

41.6.3 The 27-sheeted Covering Space

The user can check at any time whether a triangulation would be a simplicial complex in $\mathbb{T}_c^3$ and force a conversion if so. However this should be done very carefully in order to be sure that the internal structure always remains a simplicial complex and thus a triangulation.

In this example we construct a triangulation that can be converted to the 1-sheeted covering space. However, we can insert new points such that the point set does not have a Delaunay triangulation in the 1-sheeted covering space anymore, so the triangulation is not *extensible*.

```
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/Periodic_3_triangulation_traits_3.h>
#include <CGAL/Periodic_3_Delaunay_triangulation_3.h>

#include <iostream>
#include <vector>

typedef CGAL::Exact_predicates_inexact_constructions_kernel K;
typedef CGAL::Periodic_3_triangulation_traits_3<K> GT;

typedef CGAL::Periodic_3_Delaunay_triangulation_3<GT> PDT;

typedef PDT::Point                    Point;
typedef PDT::Covering_sheets          Covering_sheets;

int main()
{
  PDT T;

  // Input point grid (27 points)
  for (double x=0. ; x < .9 ; x += 0.33) {
    for (double y=0. ; y < .9 ; y += 0.33) {
      for (double z=0. ; z < .9 ; z += 0.33) {
        T.insert(Point(x,y,z));
  } } }

  Covering_sheets cs = T.number_of_sheets();
  std::cout<<"Current covering: "<<cs[0]<<' '<<cs[1]<<'
   '<<cs[2]<<std::endl;
```

```
  if ( T.is_triangulation_in_1_sheet() ) {
            // = true
    bool is_extensible = T.is_extensible_triangulation_in_1_sheet_h1()
      || T.is_extensible_triangulation_in_1_sheet_h2();
            // = false
    T.convert_to_1_sheeted_covering();
    cs = T.number_of_sheets();
    std::cout<<"Current covering: "<<cs[0]<<' '<<cs[1]<<'
   '<<cs[2]<<std::endl;
    if ( is_extensible )
            // = false
      std::cout<<"It is safe to change the triangulation
   here."<<std::endl;
    else
      std::cout<<"It is NOT safe to change the triangulation
   here!"<<std::endl;

    T.convert_to_27_sheeted_covering();
    cs = T.number_of_sheets();
    std::cout<<"Current covering: "<<cs[0]<<' '<<cs[1]<<'
   '<<cs[2]<<std::endl;
  }

  std::cout<<"It is (again) safe to modify the
   triangulation."<<std::endl;

  return 0;
}
```

File: examples/Periodic_3_triangulation_3/covering.cpp

41.6.4 Large Point Set

For large point sets there are two optimizations available. Firstly, there is spatial sorting that sorts the input points according to a Hilbert curve, see chapter 65.3 on page **??**. The second one inserts 36 appropriately chosen dummy points to avoid the use of a 27-sheeted covering space in the beginning. The 36 dummy points are deleted in the end. If the point set turns out to not have a Delaunay triangulation in the 1-sheeted covering space, the triangulation is converted to the 27-sheeted covering space during the removal of the 36 dummy points. This might take even longer than computing the triangulation without using this optimization. In general, uniformly distributed random point sets of more than 1000 points have a Delaunay triangulation in the 1-sheeted covering space.

It is recommended to run this example only when compiled in release mode because of the relatively large number of points.

```
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/Periodic_3_triangulation_traits_3.h>
#include <CGAL/Periodic_3_Delaunay_triangulation_3.h>

#include <CGAL/Random.h>
#include <CGAL/point_generators_3.h>
#include <CGAL/Timer.h>
```

```
#include <iostream>
#include <vector>

typedef CGAL::Exact_predicates_inexact_constructions_kernel K;
typedef CGAL::Periodic_3_triangulation_traits_3<K> GT;

typedef CGAL::Periodic_3_Delaunay_triangulation_3<GT> PDT;

typedef PDT::Point          Point;

int main()
{
  CGAL::Timer t;
  typedef CGAL::Creator_uniform_3<double, Point> Creator;
  CGAL::Random random(7);
  CGAL::Random_points_in_cube_3<Point, Creator> in_cube(.5, random);

  int n = 10000;
  std::vector<Point> pts;

  PDT PT1, PT2, PT3;

  // Generating n random points
  for (int i=0 ; i < n ; i++) {
    Point p = *in_cube;
    in_cube++;
    pts.push_back(Point(p.x()+.5,p.y()+.5,p.z()+.5));
  }

  // Standard insertion
  t.start();
  for (int i=0 ; i < n ; i++) {
    PT1.insert(pts[i]);
  }
  t.stop();
  std::cout<<"  Time: "<<t.time()<<" sec. (Standard
   insertion)"<<std::endl;
  t.reset();

  // Iterator range insertion using spatial sorting but no dummy points
  t.start();
  PT2.insert(pts.begin(), pts.end()); // third parameter defaults to
   false
  t.stop();
  std::cout<<"  Time: "<<t.time()<<" sec. (with spatial
   sorting)"<<std::endl;
  t.reset();

  // Iterator range insertion using spatial sorting and dummy point
   heuristic
  t.start();
  PT3.insert(pts.begin(), pts.end(), true);
  t.stop();
```

```
  std::cout<<"  Time: "<<t.time()<<" sec. (Dummy point
   heuristic)"<<std::endl;

  return 0;
}
```

File: examples/Periodic_3_triangulation_3/large_point_set.cpp

41.6.5 Geometric Access

There might be applications that need the geometric primitives of a triangulation as an input but do not require a simplicial complex. For these cases we provide the geometric iterators that return only the geometric primitives fulfilling some properties. In the following example we use the *Periodic_triangle_iterator* with the option *UNIQUE_COVER_DOMAIN*. This means that only those triangles are returned that have a non-empty intersection with the original domain of the 1-sheeted covering space, see Figure 41.4 (Page 2772). The *Periodic_triangle* is actually a three-dimensional array of point-offset pairs. We check for all three entries of the periodic triangle whether the offset is (0,0,0) using the method *is_null*. If so, we convert the periodic triangle to a *PK::Triangle_3*, which requires *exact constructions* to be exact.

```
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/Periodic_3_triangulation_traits_3.h>
#include <CGAL/Periodic_3_Delaunay_triangulation_3.h>

typedef CGAL::Exact_predicates_inexact_constructions_kernel K;
typedef CGAL::Periodic_3_triangulation_traits_3<K>          PK;
typedef CGAL::Periodic_3_Delaunay_triangulation_3<PK>       P3DT3;

typedef PK::Point_3        Point;
typedef PK::Triangle_3     Triangle;

typedef P3DT3::Periodic_triangle            Periodic_triangle;
typedef P3DT3::Periodic_triangle_iterator  Periodic_triangle_iterator;
typedef P3DT3::Iterator_type                Iterator_type;

int main() {
  P3DT3 T;

  T.insert(Point(0,0,0));
  T.insert(Point(0,0,0.5));
  T.insert(Point(0,0.5,0.5));
  T.insert(Point(0.5,0,0.5));

  Periodic_triangle pt;
  Triangle t_bd;

  // Extracting the triangles that have a non-empty intersection with
  // the original domain of the 1-sheeted covering space
  for (Periodic_triangle_iterator ptit =
   T.periodic_triangles_begin(P3DT3::UNIQUE_COVER_DOMAIN);
       ptit != T.periodic_triangles_end(P3DT3::UNIQUE_COVER_DOMAIN);
   ++ptit) {
    pt = *ptit;
```

```
    if (! (pt[0].second.is_null() && pt[1].second.is_null() &&
   pt[2].second.is_null()) ) {
      // Convert the current Periodic_triangle to a Triangle if it is
      // not strictly contained inside the original domain.
      // Note that this requires EXACT constructions to be exact!
      t_bd = T.construct_triangle(pt);
    }
  }
}
```

File: examples/Periodic_3_triangulation_3/geometric_access.cpp

41.6.6 Periodic Alpha Shapes

It is possible to use *Periodic_3_Delaunay_triangulation_3* as underlying triangulation for computing alpha shapes (cf. Chapter 43). For an example see Section 43.5.6.

41.7 Design and Implementation History

In 2006, Nico Kruithof started to work with Monique Teillaud on the 3D Periodic Triangulations package.

In 2007, Manuel Caroli continued work on the algorithms [CT09] and on the package with Monique Teillaud.

The package follows the design of the 3D Triangulations package (see Chapter 39).

3D Periodic Triangulations Reference Manual

Manuel Caroli and Monique Teillaud

The main classes of the 3D Periodic Triangulation package are *Periodic_3_triangulation_3* and *Periodic_3_Delaunay_triangulation_3*. They contain functionality to access triangulations and to run queries on them. *Periodic_3_Delaunay_triangulation_3* can construct and modify Delaunay triangulations. It takes the geometric traits as well as the triangulation data structure as template parameters.

The geometric traits class must be a model of the concept *Periodic_3DelaunayTriangulationTraits_3*. It contains all predicates and constructions that are needed by the functions in the triangulation classes.

The package uses *Triangulation_data_structure_3* to represent the triangulation. The cells and vertices need to be models of the concepts *Periodic_3TriangulationDSCellBase_3* and *Periodic_3TriangulationDSVertexBase_3*, respectively. A triangulation is stored as a collection of vertices and cells that are linked together through incidence and adjacency relations. Each cell gives access to its four incident vertices and to its four adjacent cells. Each vertex gives access to one of its incident cells.

The four vertices of a cell are indexed with 0, 1, 2 and 3 in positive orientation, the positive orientation being defined by the orientation of the underlying space $\mathbb{T}_c^3$. The neighbors of a cell are also indexed with 0, 1, 2, 3 in such a way that the neighbor indexed by i is opposite to the vertex with the same index. See Figure 41.2.

In order to be able to specify the tetrahedra that contain vertices both inside and outside the original domain we store an additional offset information in each vertex of a cell. These offsets are models of the concept *Periodic_3Offset_3*.

41.8 Classified Reference Pages

Concepts

Classes

Main Classes

Traits Classes

Enums

41.9 Alphabetical List of Reference Pages

CGAL::Periodic_3_triangulation_3<PT,TDS>

Definition

The class *Periodic_triangulation_3* represents a 3-dimensional triangulation of a point set in $\mathbb{T}_c^3$.

#include <CGAL/Periodic_3_triangulation_3.h>

Parameters

The first template argument *PT* must be a model of the *Periodic_3DelaunayTriangulationTraits_3* concept.

The second template argument *TDS* must be a model of the *TriangulationDataStructure_3* concept with some additional functionality in cells and vertices. Its default value is *Triangulation_data_structure_3< Triangulation_vertex_base_3<PT,Periodic_3_triangulation_ds_vertex_base_3<>>,Triangulation_cell_base_3< PT,Periodic_3_triangulation_ds_cell_base_3<>>>*.

Types

The class *Triangulation_3* defines the following types:

typedef PT	*Geometric_traits;*	
typedef TDS	*Triangulation_data_structure;*	
typedef Geometric_traits::Periodic_3_offset_3	*Offset;*	
typedef Geometric_traits::Iso_cuboid_3	*Iso_cuboid;*	A type representing an axis-aligned cuboid. It must be a model of *PT::Iso_cuboid_3*. Used to represent the original domain.
typedef array<int,3>	*Covering_sheets;*	Integer triple to store the number of sheets in each direction of space.
typedef Geometric_traits::Point_3	*Point;*	
typedef Geometric_traits::Segment_3	*Segment;*	
typedef Geometric_traits::Triangle_3	*Triangle;*	
typedef Geometric_traits::Tetrahedron_3	*Tetrahedron;*	
typedef std::pair< Point, Offset >	*Periodic_point;*	Represents a point-offset pair. The point in the pair lies in the original domain.
typedef array< Periodic_point, 2>	*Periodic_segment;*	
typedef array< Periodic_point, 3>	*Periodic_triangle;*	
typedef array< Periodic_point, 4>	*Periodic_tetrahedron;*	

Only vertices (0-faces) and cells (3-faces) are stored. Edges (1-faces) and facets (2-faces) are not explicitly represented and thus there are no corresponding classes (see Section 41.2).

typedef Triangulation_data_structure::Vertex	*Vertex;*
typedef Triangulation_data_structure::Cell	*Cell;*
typedef Triangulation_data_structure::Edge	*Edge;*

typedef Triangulation_data_structure::Facet *Facet;*

The vertices and faces of the triangulations are accessed through *handles*, *iterators* and *circulators*. A handle is a type which supports the two dereference operators *operator** and *operator->*. The Handle concept is documented in the support library. Iterators and circulators are bidirectional and non-mutable. The edges and facets of the triangulation can also be visited through iterators and circulators which are bidirectional and non-mutable.

Iterators and circulators are convertible to the corresponding handles, thus the user can pass them directly as arguments to the functions.

typedef Triangulation_data_structure::Vertex_handle	*Vertex_handle;*	handle to a vertex
typedef Triangulation_data_structure::Cell_handle	*Cell_handle;*	handle to a cell
typedef Triangulation_data_structure::size_type	*size_type;*	Size type (an unsigned integral type)
typedef Triangulation_data_structure::difference_type	*difference_type;*	Difference type (a signed integral type)
typedef Triangulation_data_structure::Cell_iterator	*Cell_iterator;*	iterator over cells
typedef Triangulation_data_structure::Facet_iterator	*Facet_iterator;*	iterator over facets
typedef Triangulation_data_structure::Edge_iterator	*Edge_iterator;*	iterator over edges
typedef Triangulation_data_structure::Vertex_iterator	*Vertex_iterator;*	iterator over vertices
Periodic_3_triangulation_3<PT,TDS>:: Unique_vertex_iterator		iterator over the vertices whose corresponding points lie in the original domain, i.e. for each set of periodic copies the *Unique_vertex_iterator* iterates over exactly one representative.
typedef Triangulation_data_structure::Cell_circulator	*Cell_circulator;*	circulator over all cells incident to a given edge
typedef Triangulation_data_structure::Facet_circulator	*Facet_circulator;*	circulator over all facets incident to a given edge

Geometric iterators:

Periodic_3_triangulation_3<PT,TDS>:: Periodic_tetrahedron_iterator	iterator over the tetrahedra corresponding to cells of the triangulation.
Periodic_3_triangulation_3<PT,TDS>:: Periodic_triangle_iterator	iterator over the triangles corresponding to facets of the triangulation.
Periodic_3_triangulation_3<PT,TDS>:: Periodic_segment_iterator	iterator over the segments corresponding to edges of the triangulation.
Periodic_3_triangulation_3<PT,TDS>:: Periodic_point_iterator	iterator over the points corresponding to vertices of the triangulation.

Enums:

The triangulation class also defines the following enum types:

To specify which case occurs when locating a point in the triangulation.
enum Locate_type { *VERTEX=0, EDGE, FACET, CELL, EMPTY*}*;*

To specify the behavior of geometric iterators.
enum Iterator_type { *STORED=0, UNIQUE, STORED_COVER_DOMAIN, UNIQUE_COVER_DOMAIN*}*;*

Creation

Periodic_3_triangulation_3<PT,TDS> t(Iso_cuboid domain = Iso_cuboid(0,0,0,1,1,1), Geometric_traits traits = Geometric_traits())

Introduces an empty triangulation *t* with *domain* as original domain.
Precondition: *domain* is a cube.

Periodic_3_triangulation_3<PT,TDS> t(Periodic_3_triangulation_3 tr);

Copy constructor. All vertices and faces are duplicated.

Assignment

Periodic_3_triangulation_3 & *t = Periodic_3_triangulation_3 tr*

The triangulation *tr* is duplicated, and modifying the copy after the duplication does not modify the original. The previous triangulation held by *t* is deleted.

void *t.swap(Periodic_3_triangulation_3 & tr)*

The triangulations *tr* and *t* are swapped. *t.swap(tr)* should be preferred to *t = tr* or to *t(tr)* if *tr* is deleted after that. Indeed, there is no copy of cells and vertices, thus this method runs in constant time.

void *t.clear()* Deletes all vertices and all cells of *t*.

template < class PT, class TDS1, class TDS2 >
bool *operator==(Periodic_3_triangulation_3<PT, TDS1> t1, Periodic_3_triangulation_3<PT, TDS2> t2)*

Equality operator. Returns true iff there exist a bijection between the vertices of *t1* and those of *t2* and a bijection between the cells of *t1* and those of *t2*, which preserve the geometry of the triangulation, that is, the points of each corresponding pair of vertices are equal, and the tetrahedra corresponding to each pair of cells are equal (up to a permutation of their vertices).

template < class PT, class TDS1, class TDS2 >

bool *operator!=(Periodic_3_triangulation_3<PT, TDS1> t1, Periodic_3_triangulation_3<PT, TDS2> t2)*

The opposite of *operator==*.

Access Functions

Geometric_traits	*t.geom_traits() const*	Returns a const reference to the geometric traits object.
Triangulation_data_structure	*t.tds() const*	Returns a const reference to the triangulation data structure.
Iso_cuboid	*t.domain() const*	Returns the original domain.

advanced

Covering_sheets *t.number_of_sheets() const*

Returns the number of sheets of the covering space the triangulation is currently computed in.

Non const access

void *t.set_domain(const Iso_cuboid dom)*

Changes the domain. Note that this function calls *clear()*, i.e., it erases the existing triangulation.

The responsibility of keeping a valid triangulation belongs to the user when using advanced operations allowing a direct manipulation of the *tds*. This method is mainly a help for users implementing their own triangulation algorithms.

Triangulation_data_structure& *t.tds()* Returns a reference to the triangulation data structure.

Non-constant-time queries and conversions

bool *t.is_extensible_triangulation_in_1_sheet_h1() const*

The current triangulation remains a triangulation in the 1-sheeted covering space even after adding points if this method returns *true*. This test relies on a heuristic, i.e. if it answers *false* nothing is known. This test runs in constant-time when not computing in the 1-sheeted covering space. (This test uses the length of the longest edge in the triangulation as a criterion [CT09].)

bool *t.is_extensible_triangulation_in_1_sheet_h2() const*

The same as *is_extensible_triangulation_in_1_sheet_h1()* but with a more precise heuristic, i.e. it might answer *true* in cases in which *is_extensible_triangulation_in_1_sheet_h1()* would not. However, it is much less time efficient when not computing in the 1-sheeted covering space. (This test uses the diameter of the largest empty ball in the input point set as a criterion [CT09].)

bool *t.is_triangulation_in_1_sheet() const* Returns *true* if the current triangulation would still be a triangulation in the 1-sheeted covering space, returns *false* otherwise.

It is not recommended to interfere with the built-in covering management. Especially a premature conversion to the 1-sheeted covering space might lead to problems when modifying the triangulation later.

void *t.convert_to_1_sheeted_covering() const* Converts the current triangulation into the same periodic triangulation in the 1-sheeted covering space.

void *t.convert_to_27_sheeted_covering() const* Converts the current triangulation into the same periodic triangulation in the 27-sheeted covering space.

advanced

size_type *t.number_of_vertices() const* Returns the number of vertices. Counts all vertices that are representatives of the same point in $\mathbb{T}_c^3$ as one vertex.

size_type *t.number_of_cells() const* Returns the number of cells. Counts all cells that are representatives of the same tetrahedron in $\mathbb{T}_c^3$ as one cell.

advanced

size_type *t.number_of_stored_vertices() const*

Returns the number of vertices in the data structure. This is the same as the number of sheets times *number_of_vertices()*.

size_type *t.number_of_stored_cells() const*

Returns the number of cells in the data structure. This is the same as the number of sheets times *number_of_cells()*.

advanced

Non-constant-time access functions

size_type *t.number_of_edges() const* Returns the number of edges. Counts all edges that are representatives of the same segment in $\mathbb{T}_c^3$ as one edge.

size_type *t.number_of_facets() const* Returns the number of facets. Counts all facets that are representatives of the same triangle in $\mathbb{T}_c^3$ as one facet.

advanced

size_type *t.number_of_stored_edges() const*

Returns the number of edges in the data structure. This is the same as the number of sheets times *number_of_edges()*.

size_type *t.number_of_stored_facets() const*

Returns the number of facets in the data structure. This is the same as the number of sheets times *number_of_facets()*.

advanced

Geometric access functions

Periodic_point *t.periodic_point(const Vertex_handle v) const*

Returns the periodic point given by vertex *v*. If *t* is represented in the 1-sheeted covering space, the offset is always zero. Otherwise *v* can correspond to a periodic copy outside *domain* of an input point.

Periodic_point *t.periodic_point(const Cell_handle c, int i) const*

If *t* is represented in the 1-sheeted covering space, this function returns the periodic point given by the *i*-th vertex of cell *c*, that is the point in the original domain and the offset of the vertex in *c*. If *t* is represented in the 27-sheeted covering space, this offset is possibly added to another offset determining the periodic copy.
Precondition: $i \in \{0,1,2,3\}$

Periodic_segment *t.periodic_segment(const Cell_handle c, int i, int j) const*

Returns the periodic segment formed by the two point-offset pairs corresponding to the two vertices of edge *(c,i,j)*.
Precondition: $i, j \in \{0,1,2,3\}$, $i \neq j$

Periodic_segment *t.periodic_segment(Edge e) const*

Same as the previous method for edge *e*.

Periodic_triangle *t.periodic_triangle(const Cell_handle c, int i) const*

Returns the periodic triangle formed by the three point-offset pairs corresponding to the three vertices of facet *(c,i)*. The triangle is oriented so that its normal points to the inside of cell *c*.
Precondition: $i \in \{0,1,2,3\}$

Periodic_triangle *t.periodic_triangle(Facet f) const*

Same as the previous method for facet *f*.

Periodic_tetrahedron *t.periodic_tetrahedron(const Cell_handle c) const*

Returns the periodic tetrahedron formed by the four point-offset pairs corresponding to the four vertices of *c*.

Note that a traits class providing exact constructions should be used in order to guarantee the following operations to be exact (as opposed to computing the triangulation only, which requires only exact predicates).

Point *t.point(Periodic_point p) const*

Converts the *Periodic_point s* (point-offset pair) to the corresponding *Point* in $\mathbb{R}^3$.

Segment *t.segment(Periodic_segment s) const*

Converts the *Periodic_segment s* to a *Segment*.

Triangle *t.triangle(Periodic_triangle t) const*

Converts the *Periodic_triangle t* to a *Triangle*.

Tetrahedron *t.tetrahedron(Periodic_tetrahedron t) const*

Converts the *Periodic_tetrahedron t* to a *Tetrahedron*.

Queries

bool *t.is_vertex(Point p, Vertex_handle & v) const*

Tests whether *p* is a vertex of *t* by locating *p* in the triangulation. If *p* is found, the associated vertex *v* is given.

bool *t.is_vertex(Vertex_handle v) const*

Tests whether *v* is a vertex of *t*.

bool *t.is_edge(Vertex_handle u, Vertex_handle v, Cell_handle & c, int & i, int & j) const*

Tests whether *(u,v)* is an edge of *t*. If the edge is found, it gives a cell *c* having this edge and the indices *i* and *j* of the vertices *u* and *v* in *c*, in this order.
Precondition: *u* and *v* are vertices of *t*.

bool *t.is_edge(Vertex_handle u,*
Offset offu,
Vertex_handle v,
Offset offv,
Cell_handle & c,
int & i,
int & j) const

Tests whether *((u,offu),(v,offu))* is an edge of *t*. If the edge is found, it gives a cell *c* having this edge and the indices *i* and *j* of the vertices *u* and *v* in *c*, in this order.
Precondition: *u* and *v* are vertices of *t*.

bool *t.is_facet(Vertex_handle u,*
Vertex_handle v,
Vertex_handle w,
Cell_handle & c,
int & i,
int & j,
int & k) const

Tests whether *(u,v,w)* is a facet of *t*. If the facet is found, it computes a cell *c* having this facet and the indices *i*, *j* and *k* of the vertices *u*, *v* and *w* in *c*, in this order.
Precondition: *u*, *v* and *w* are vertices of *t*.

bool *t.is_facet(Vertex_handle u,*
Offset offu,
Vertex_handle v,
Offset offv,
Vertex_handle w,
Offset offw,
Cell_handle & c,
int & i,
int & j,
int & k) const

Tests whether *((u,offu),(v,offv),(w,offw))* is a facet of *t*. If the facet is found, it computes a cell *c* having this facet and the indices *i*, *j* and *k* of the vertices *u*, *v* and *w* in *c*, in this order.
Precondition: *u*, *v* and *w* are vertices of *t*.

bool *t.is_cell(Cell_handle c) const*

Tests whether *c* is a cell of *t*.

bool *t.is_cell(Vertex_handle u,*
Vertex_handle v,
Vertex_handle w,
Vertex_handle x,
Cell_handle & c,
int & i,
int & j,
int & k,
int & l) const

Tests whether *(u,v,w,x)* is a cell of *t*. If the cell *c* is found, the method computes the indices *i*, *j*, *k* and *l* of the vertices *u*, *v*, *w* and *x* in *c*, in this order.
Precondition: *u*, *v*, *w* and *x* are vertices of *t*.

bool *t.is_cell(Vertex_handle u, Vertex_handle v, Vertex_handle w, Vertex_handle x, Cell_handle & c) const*

Tests whether *(u,v,w,x)* is a cell of *t* and computes this cell *c*.
Precondition: *u*, *v*, *w* and *x* are vertices of *t*.

bool *t.is_cell(Vertex_handle u,*
Offset offu,
Vertex_handle v,
Offset offv,
Vertex_handle w,
Offset offw,
Vertex_handle x,
Offset offx,
Cell_handle & c,
int & i,
int & j,
int & k,

int & l) const

Tests whether *((u,offu),(v,offv),(w,offv),(x,offx))* is a cell of *t*. If the cell *c* is found, the method computes the indices *i*, *j*, *k* and *l* of the vertices *u*, *v*, *w* and *x* in *c*, in this order.
Precondition: *u*, *v*, *w* and *x* are vertices of *t*.

bool *t.is_cell(Vertex_handle u,*
Offset offu,
Vertex_handle v,
Offset offv,
Vertex_handle w,
Offset offw,
Vertex_handle x,
Offset offx,
Cell_handle & c) const

Tests whether *((u,offu),(v,offv),(w,offv),(x,offx))* is a cell of *t* and computes this cell *c*.
Precondition: *u*, *v*, *w* and *x* are vertices of *t*.

There is a method *has_vertex* in the cell class. The analogous methods for facets are defined here.

bool *t.has_vertex(Facet f, Vertex_handle v, int & j) const*

If *v* is a vertex of *f*, then *j* is the index of *v* in the cell *f.first*, and the method returns *true*.

bool *t.has_vertex(Cell_handle c, int i, Vertex_handle v, int & j) const*

Same for facet *(c,i)*. Computes the index *j* of *v* in *c*.

bool *t.has_vertex(Facet f, Vertex_handle v) const*
bool *t.has_vertex(Cell_handle c, int i, Vertex_handle v) const*

Same as the first two methods, but these two methods do not return the index of the vertex.

The following three methods test whether two facets have the same vertices.

bool *t.are_equal(Cell_handle c, int i, Cell_handle n, int j) const*
bool *t.are_equal(Facet f, Facet g) const*
bool *t.are_equal(Facet f, Cell_handle n, int j) const*

Point location

The class *Periodic_3_triangulation_3<PT,TDS>* provides three functions to locate a given point with respect to a triangulation. It provides also functions to test if a given point is inside a face or not. Note that the class *Periodic_3_Delaunay_triangulation_3* also provides a *nearest_vertex()* function.

Cell_handle *t.locate(Point query, Cell_handle start = Cell_handle())*

Returns the cell that contains the query in its interior. If *query* lies on a facet, an edge or on a vertex, one of the cells having *query* on its boundary is returned.
The optional argument *start* is used as a starting place for the search.
Precondition: *query* lies in the original domain *domain*.

Cell_handle *t.locate(Point query, Locate_type & lt, int & li, int & lj, Cell_handle start = Cell_handle())*

The k-face that contains *query* in its interior is returned, by means of the cell returned together with *lt*, which is set to the locate type of the query (*VERTEX, EDGE, FACET, CELL*) and two indices *li* and *lj* that specify the k-face of the cell containing *query*.
If the k-face is a cell, *li* and *lj* have no meaning; if it is a facet (resp. vertex), *li* gives the index of the facet (resp. vertex) and *lj* has no meaning; if it is an edge, *li* and *lj* give the indices of its vertices.
If there is no vertex in the triangulation yet, *lt* is set to *EMPTY* and *locate* returns the default constructed handle.
The optional argument *start* is used as a starting place for the search.
Precondition: *query* lies in the original domain *domain*.

Bounded_side *t.side_of_cell(Point p, Cell_handle c, Locate_type & lt, int & li, int & lj) const*

Returns a value indicating on which side of the oriented boundary of c the point p lies. More precisely, it returns:
- *ON_BOUNDED_SIDE* if p is inside the cell.
- *ON_BOUNDARY* if p on the boundary of the cell. Then *lt* together with *li* and *lj* give the precise location on the boundary. (See the descriptions of the *locate* methods.)
- *ON_UNBOUNDED_SIDE* if p lies outside the cell.
Precondition: *query* lies in the original domain *domain*.

Traversal of the Triangulation

The periodic triangulation class provides several iterators and circulators that allow one to traverse it.

Cell, Face, Edge and Vertex Iterators

The following iterators allow the user to visit cells, facets, edges and vertices of the stored triangulation, i.e. in case of computing in a multiply sheeted covering space all stored periodic copies of each item are returned. These iterators are non-mutable, bidirectional and their value types are respectively *Cell*, *Facet*, *Edge* and *Vertex*. They are all invalidated by any change in the triangulation.

Vertex_iterator	*t.vertices_begin() const*	Starts at an arbitrary vertex. Iterates over all vertices. Returns *vertices_end()* if *t.number_of_vertices()* $= 0$.
Vertex_iterator	*t.vertices_end() const*	Past-the-end iterator
Edge_iterator	*t.edges_begin() const*	Starts at an arbitrary edge. Iterates over all edges. Returns *edges_end()* if *t.number_of_vertices()* $= 0$.
Edge_iterator	*t.edges_end() const*	Past-the-end iterator
Facet_iterator	*t.facets_begin() const*	Starts at an arbitrary facet. Iterates over all facets. Returns *facets_end()* if *t.number_of_vertices()* $= 0$.
Facet_iterator	*t.facets_end() const*	Past-the-end iterator
Cell_iterator	*t.cells_begin() const*	Starts at an arbitrary cell. Iterates over all cells. Returns *cells_end()* if *t.number_of_vertices()* $= 0$.

Cell_iterator	*t.cells_end() const*	Past-the-end iterator
Unique_vertex_iterator	*t.unique_vertices_begin() const*	Starts at an arbitrary vertex. Iterates over all vertices whose corresponding points lie in the original domain, i.e. for each set of periodic copies the *Unique_vertex_iterator* iterates over exactly one representative. Returns *unique_vertices_end()* if *t.number_of_vertices()* $= 0$.
Unique_vertex_iterator	*t.unique_vertices_end() const*	Past-the-end iterator

Geometric iterators

The following iterators allow the user to obtain geometric primitives corresponding to cells, facets, edges, and vertices of the triangulation. These iterators are non-mutable, bidirectional and their value types are respectively *Periodic_point*, *Periodic_segment*, *Periodic_triangle*, and *Periodic_tetrahedron*. They are all invalidated by any change in the triangulation. If the periodic triangulation is not computed in the 1-sheeted covering space, these iterators can be used to retain only the geometric primitives in the original domain. This can be controlled using the enum *Iterator_type*, see page 2796.

Periodic_point_iterator	*t.periodic_points_begin(Iterator_type it = STORED) const*	Iterates over the points of the triangulation. Its behavior is defined by the *Iterator_type it* as described on page 2796.
Periodic_point_iterator	*t.periodic_points_end(Iterator_type it = STORED) const*	Past-the-end iterator. Note that to match another *Periodic_point_iterator* both must have the same *Iterator_type it*.
Periodic_segment_iterator	*t.periodic_segments_begin(Iterator_type it = STORED) const*	Iterates over the segments of the triangulation. Its behavior is defined by the *Iterator_type it* as described on page 2796.
Periodic_segment_iterator	*t.periodic_segments_end(Iterator_type it = STORED) const*	Past-the-end iterator. Note that to match another *Periodic_segment_iterator* both must have the same *Iterator_type it*.
Periodic_triangle_iterator	*t.periodic_triangles_begin(Iterator_type it = STORED) const*	Iterates over the triangles of the triangulation. Its behavior is defined by the *Iterator_type it* as described on page 2796.
Periodic_triangle_iterator	*t.periodic_triangles_end(Iterator_type it = STORED) const*	Past-the-end iterator. Note that to match another *Periodic_triangle_iterator* both must have the same *Iterator_type it*.

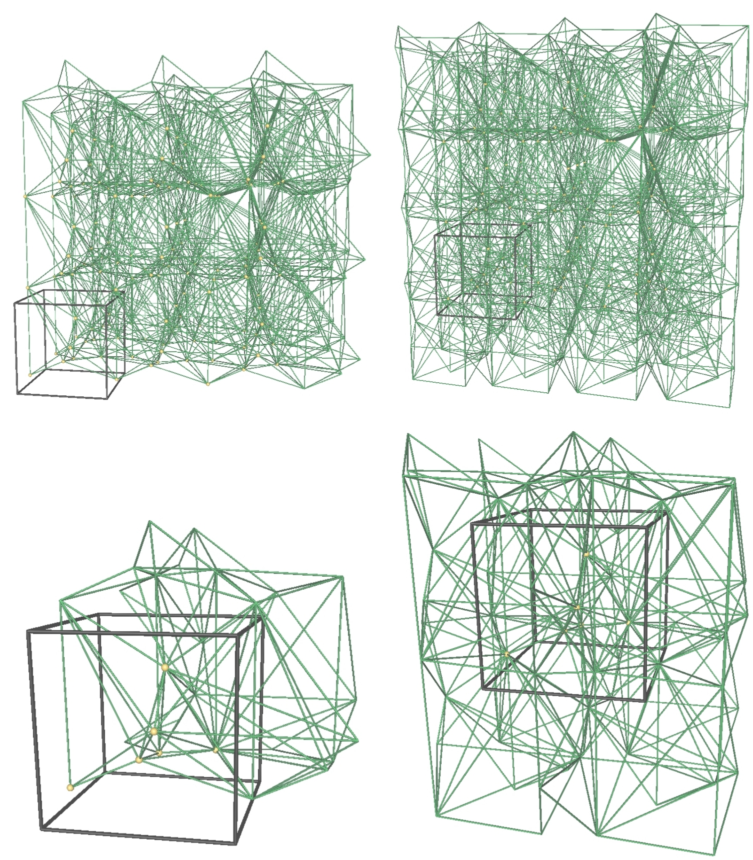


Figure 41.4: The four different modes of the geometric iterators: STORED, STORED_COVER_DOMAIN, UNIQUE, UNIQUE_COVER_DOMAIN. Note that in case of computing in the 1-sheeted covering space, STORED and UNIQUE give the same result.

Periodic_tetrahedron_iterator *t.periodic_tetrahedra_begin(Iterator_type it = STORED) const*

Iterates over the tetrahedra of the triangulation. Its behavior is defined by the *Iterator_type it* as described on page 2796.

Periodic_tetrahedron_iterator *t.periodic_tetrahedra_end(Iterator_type it = STORED) const*

Past-the-end iterator. Note that to match another *Periodic_tetrahedron_iterator* both must have the same *Iterator_type it*.

Cell and Facet Circulators

The following circulators respectively visit all cells or all facets incident to a given edge. They are non-mutable and bidirectional. They are invalidated by any modification of one of the cells traversed.

Cell_circulator *t.incident_cells(Edge e) const* Starts at an arbitrary cell incident to *e*.

Cell_circulator *t.incident_cells(Cell_handle c, int i, int j) const*

As above for edge *(i,j)* of *c*.

Cell_circulator *t.incident_cells(Edge e, Cell_handle start) const*

Starts at cell *start*.
Precondition: *start* is incident to *e*.

Cell_circulator *t.incident_cells(Cell_handle c, int i, int j, Cell_handle start) const*

As above for edge *(i,j)* of *c*.

Facet_circulator *t.incident_facets(Edge e) const* Starts at an arbitrary facet incident to *e*.

Facet_circulator *t.incident_facets(Cell_handle c, int i, int j) const*

As above for edge *(i,j)* of *c*.

Facet_circulator *t.incident_facets(Edge e, Facet start) const*

Starts at facet *start*.
Precondition: *start* is incident to *e*.

Facet_circulator *t.incident_facets(Edge e, Cell_handle start, int f) const*

Starts at facet of index *f* in *start*.

Facet_circulator *t.incident_facets(Cell_handle c, int i, int j, Facet start) const*

As above for edge *(i,j)* of *c*.

Facet_circulator *t.incident_facets(Cell_handle c, int i, int j, Cell_handle start, int f) const*

As above for edge *(i,j)* of *c* and facet *(start,f)*.

Traversal of the incident cells and facets, and the adjacent vertices of a given vertex

template <class OutputIterator>

OutputIterator *t.incident_cells(Vertex_handle v, OutputIterator cells) const*

Copies the *Cell_handle*s of all cells incident to *v* to the output iterator *cells*. Returns the resulting output iterator.
Precondition: $v \neq$ *Vertex_handle()*, *t.is_vertex(v)*.

template <class OutputIterator>
OutputIterator *t.incident_facets(Vertex_handle v, OutputIterator facets) const*

Copies the *Facet*s incident to *v* to the output iterator *facets*. Returns the resulting output iterator.
Precondition: $v \neq$ *Vertex_handle()*, *t.is_vertex(v)*.

template <class OutputIterator>
OutputIterator *t.incident_edges(Vertex_handle v, OutputIterator edges) const*

Copies the *Edge*s incident to *v* to the output iterator *edges*. Returns the resulting output iterator.
Precondition: $v \neq$ *Vertex_handle()*, *t.is_vertex(v)*.

template <class OutputIterator>
OutputIterator *t.adjacent_vertices(Vertex_handle v, OutputIterator vertices) const*

Copies the *Vertex_handle*s of all vertices adjacent to *v* to the output iterator *vertices*. Returns the resulting output iterator.
Precondition: $v \neq$ *Vertex_handle()*, *t.is_vertex(v)*.

size_type *t.degree(Vertex_handle v) const*

Returns the degree of a vertex, that is, the number of adjacent vertices.
Precondition: $v \neq$ *Vertex_handle()*, *t.is_vertex(v)*.

Traversal between adjacent cells

int *t.mirror_index(Cell_handle c, int i) const*

Returns the index of *c* in its i^{th} neighbor.
Precondition: $i \in \{0,1,2,3\}$.

Vertex_handle *t.mirror_vertex(Cell_handle c, int i) const*

Returns the vertex of the i^{th} neighbor of *c* that is opposite to *c*.
Precondition: $i \in \{0,1,2,3\}$.

Facet *t.mirror_facet(Facet f) const* Returns the same facet viewed from the other adjacent cell.

advanced

Checking

The responsibility of keeping a valid triangulation belongs to the user when using advanced operations allowing a direct manipulation of cells and vertices. We provide the user with the following methods to help debugging.

bool *t.is_valid(bool verbose = false) const*

Checks the combinatorial validity of the triangulation. Checks also the validity of its geometric embedding (see Section 41.2).
When *verbose* is set to true, messages describing the first invalidity encountered are printed.

bool *t.is_valid(Cell_handle c, bool verbose = false) const*

Checks the combinatorial validity of the cell by calling the *is_valid* method of the *Triangulation_data_structure* cell class. Also checks the geometric validity of *c*, if *c* is finite. (See Section 41.2.)
When *verbose* is set to *true*, messages are printed to give a precise indication of the kind of invalidity encountered.

advanced

I/O

istream&	*istream& is >> Periodic_3_triangulation_3 &t*	Reads a triangulation from *is* and stores it in *t*. *Precondition*: *is* has the below described format.
ostream&	*ostream& os << Periodic_3_triangulation_3 t*	Writes the triangulation *t* into *os*.

The information in the *iostream* is:

- the original domain
- the number of sheets of the covering space as in *number_of_sheets()*
- the number of vertices
- the non-combinatorial information of vertices (point resp. point-offset pairs, etc.)
- the number of cells
- the indices of the vertices of each cell
- the indices of the neighbors of each cell, where the index corresponds to the preceding list of cells
- the offsets corresponding to the vertices of the cells
- the non-combinatorial information of each cell

See Also

Periodic_3_Delaunay_triangulation_3

CGAL::Periodic_3_Delaunay_triangulation_3<PT,TDS>

Definition

The class *Periodic_3_Delaunay_triangulation_3* represents a Delaunay triangulation in three-dimensional periodic space.

#include <CGAL/Periodic_3_Delaunay_triangulation_3.h>

Parameters

The first template argument *PT* must be a model of the *Periodic_3DelaunayTriangulationTraits_3* concept.

The second template argument *TDS* must be a model of the *TriangulationDataStructure_3* concept with some additional functionality in cells and vertices. Its default value is *Triangulation_data_structure_3< Triangulation_vertex_base_3<PT,Periodic_3_triangulation_ds_vertex_base_3<>>,Triangulation_cell_base_3< PT,Periodic_3_triangulation_ds_cell_base_3<>>>*.

Inherits From

Periodic_3_triangulation_3<Periodic_3DelaunayTriangulationTraits_3,TriangulationDataStructure_3>

Creation

Periodic_3_Delaunay_triangulation_3<PT,TDS> dt(Iso_cuboid domain = Iso_cuboid(0,0,0,1,1,1),
Geom_traits traits = Geom_traits())

Creates an empty periodic Delaunay triangulation *dt*, with *domain* as original domain and possibly specifying a traits class *traits*.
Precondition: *domain* is a cube.

Periodic_3_Delaunay_triangulation_3<PT,TDS> dt(Periodic_3_Delaunay_triangulation_3 dt1);

Copy constructor.

template < class InputIterator >
Periodic_3_Delaunay_triangulation_3<PT,TDS> dt(InputIterator first,
InputIterator last,
Iso_cuboid domain = Iso_cuboid(0,0,0,1,1,1),
Geom_traits traits = Geom_traits())

Equivalent to constructing an empty triangulation with the optional domain and traits class arguments and calling *insert(first,last)*.
Precondition: The *value_type* of *first* and *last* are *Point*s lying inside *domain* and *domain* is a cube.

Operations

Insertion

The following methods insert points in the triangulation ensuring the empty sphere property of Delaunay triangulations. The inserted points need to lie in the original domain (see Section 41.1 of the user manual).

In the degenerate case when there are co-spherical points, the Delaunay triangulation is known not to be uniquely defined. In this case, CGAL chooses a particular Delaunay triangulation using a symbolic perturbation scheme [DT03].

Note that insertion of a new point can cause a switch from computing in the 27-sheeted covering space to computing in the 1-sheeted covering space, which invalidates some *Vertex_handle*s and *Cell_handle*s.

Vertex_handle *dt.insert(Point p, Cell_handle start = Cell_handle())*

Inserts point *p* in the triangulation and returns the corresponding vertex. The optional argument *start* is used as a starting place for the point location.
Precondition: *p* lies in the original domain *domain*.

Vertex_handle *dt.insert(Point p, Locate_type lt, Cell_handle loc, int li, int lj)*

Inserts point *p* in the triangulation and returns the corresponding vertex. Similar to the above *insert()* function, but takes as additional parameter the return values of a previous location query. See description of *Periodic_3_triangulation_3::locate()*.
Precondition: *p* lies in the original domain *domain*.

The following method allows one to insert several points. It returns the number of inserted points.

template < class InputIterator >
std::ptrdiff_t *dt.insert(InputIterator first, InputIterator last, bool is_large_point_set = false)*

Inserts the points in the iterator range $[first, last)$. Returns the number of inserted points. This function uses spatial sorting (cf. chapter 65.3) and therefore is not guaranteed to insert the points following the order of *InputIterator*. If the third argument *is_large_point_set* is set to *true* a heuristic for optimizing the insertion of large point sets is applied.
Precondition: The *value_type* of *first* and *last* are *Point*s lying inside *domain*.

Point moving

Vertex_handle *dt.move_point(Vertex_handle v, Point p)*

Moves the point stored in *v* to *p*, while preserving the Delaunay property. This performs an action semantically equivalent to *remove(v)* followed by *insert(p)*, but is supposedly faster when the point has not moved much. Returns the handle to the new vertex.
Precondition: *p* lies in the original domain *domain*.

Removal

When a vertex *v* is removed from a triangulation, all the cells incident to *v* must be removed, and the polyhedral region consisting of all the tetrahedra that are incident to *v* must be re-triangulated. So, the problem reduces to triangulating a polyhedral region, while preserving its boundary, or to compute a *constrained* triangulation. This is known to be sometimes impossible: the Schönhardt polyhedron cannot be triangulated [She98a].

However, when dealing with Delaunay triangulations, the case of such polyhedra that cannot be re-triangulated cannot happen, so CGAL proposes a vertex removal.

void *dt.remove(Vertex_handle v)*

Removes the vertex *v* from the triangulation. When computing in the 27-sheeted covering space it removes all 27 copies of *v*.

template < class InputIterator >
std::ptrdiff_t *dt.remove(InputIterator first, InputIterator beyond)*

Removes the vertices specified by the iterator range (*first, beyond*) of value type *Vertex_handle*. *remove()* is called for each element of the range. The number of vertices removed is returned; this number does not account for periodic copies of removed vertices.
Precondition: The iterator must not iterate over several periodic copies of the same vertex, use e.g. the *Unique_vertex_iterator*.

Queries

Bounded_side *dt.side_of_sphere(Cell_handle c, Point p, Offset off = Offset(0,0,0))*

Returns a value indicating on which side of the circumscribed sphere of *c* the point-offset pair (*p*,*off*) lies. More precisely, it returns:
- *ON_BOUNDED_SIDE* if (*p*,*off*) is inside the sphere.
- *ON_BOUNDARY* if (*p*,*off*) on the boundary of the sphere.
- *ON_UNBOUNDED_SIDE* if (*p*,*off*) lies outside the sphere.
Precondition: *p* lies in the original domain *domain*.

Vertex_handle *dt.nearest_vertex(Point p, Cell_handle c = Cell_handle())*

Returns any nearest vertex to the point *p*, or the default constructed handle if the triangulation is empty. The optional argument *c* is a hint specifying where to start the search. It always returns a vertex corresponding to a point inside *domain* even if computing in a multiply sheeted covering space.
Precondition: *c* is a cell of *dt* and *p* lies in the original domain *domain*.

Vertex_handle *dt.nearest_vertex_in_cell(Cell_handle c, Point p, Offset off = Offset(0,0,0))*

Returns the vertex of the cell *c* that is nearest to the point-offset pair (*p*,*off*).
Precondition: *p* lies in the original domain *domain*.

A point-offset pair (*p*,*off*) is said to be in conflict with a cell *c* iff *dt.side_of_sphere(c, p, off)* returns *ON_BOUNDED_SIDE*. The set of cells that are in conflict with (*p*,*off*) is star-shaped.

template <class OutputIteratorBoundaryFacets, class OutputIteratorCells, class OutputIteratorInternalFacets>

Triple<OutputIteratorBoundaryFacets, OutputIteratorCells, OutputIteratorInternalFacets>

dt.find_conflicts(Point p,
Cell_handle c,
OutputIteratorBoundaryFacets bfit,
OutputIteratorCells cit,
OutputIteratorInternalFacets ifit)

Computes the conflict hole induced by *p*. The starting cell *c* must be in conflict. Then this function returns respectively in the output iterators:
– *cit*: the cells in conflict.
– *bfit*: the facets on the boundary, that is, the facets *(t, i)* where the cell *t* is in conflict, but *t->neighbor(i)* is not.
– *ifit*: the facets inside the hole, that is, delimiting two cells in conflict.
Returns the pair composed of the resulting output iterators.
Precondition: *c* is in conflict with *p* and *p* lies in the original domain *domain*.

template <class OutputIterator>
OutputIterator

dt.vertices_in_conflict(Point p, Cell_handle c, OutputIterator res)

Similar to *find_conflicts()*, but reports the vertices which are on the boundary of the conflict hole of *p*, in the output iterator *res*. Returns the resulting output iterator.
Precondition: *c* is in conflict with *p* and *p* lies in the original domain *domain*.

A face (cell, facet or edge) is said to be a Gabriel face iff its smallest circumscribing sphere do not enclose any vertex of the triangulation. Any Gabriel face belongs to the Delaunay triangulation, but the reciprocal is not true. The following member functions test the Gabriel property of Delaunay faces.

bool	*dt.is_Gabriel(Cell_handle c, int i)*
bool	*dt.is_Gabriel(Cell_handle c, int i, int j)*
bool	*dt.is_Gabriel(Facet f)*
bool	*dt.is_Gabriel(Edge e)*

Voronoi diagram

CGAL offers several functions to display the Voronoi diagram of a set of points in 3D.

Note that a traits class providing exact constructions should be used in order to guarantee the computation of the Voronoi diagram (as opposed to computing the triangulation only, which requires only exact predicates).

Point	*dt.dual(Cell_handle c) const*	Returns the representative of the circumcenter of the four vertices of c that lies in the original domain *domain*.
Periodic_segment	*dt.dual(Facet f) const*	Returns the dual of facet *f*, which is a periodic segment.
Periodic_segment	*dt.dual(Cell_handle c, int i) const*	

same as the previous method for facet *(c,i)*.
Precondition: $i \in \{0,1,2,3\}$

template <class OutputIterator>
OutputIterator *dt.dual(Edge e, OutputIterator pts) const*

Returns in the output iterator the points of the dual polygon of edge *e* in the same order as the *Facet_circulator* returns facets incident to the edge *e*. The points form the dual polygon in $\mathbb{R}^3$, so they do not necessarily all lie inside the original domain.

template <class OutputIterator>
OutputIterator *dt.dual(Cell_handle c, int i, int j, OutputIterator pts) const*

same as the previous method for edge *(c,i,j)*.
Precondition: $i, j \in \{0,1,2,3\}, i \neq j$

template <class OutputIterator>
OutputIterator *dt.dual(Vertex_handle v, OutputIterator pts) const*

Returns in the output iterator the points of the dual polyhedron of vertex *v* in no particular order. The points form the dual polyhedron in $\mathbb{R}^3$, so they do not necessarily lie all inside the original domain.

template <class Stream>
Stream& *dt.draw_dual(Stream & os)* Sends the set of duals to all the facets of *dt* into *os*.

Geom_traits::FT *dt.dual_volume(Vertex_handle v) const*

Returns the volume of the Voronoi cell dual to *v*.

Point *dt.dual_centroid(Vertex_handle v) const*

Returns the centroid of the Voronoi cell dual to *v*.

advanced

Checking

bool *dt.is_valid(bool verbose = false) const*

Checks the combinatorial validity of the triangulation and the validity of its geometric embedding (see Section 41.2). Also checks that all the circumscribing spheres of cells are empty.
When *verbose* is set to true, messages describing the first invalidity encountered are printed.

bool *dt.is_valid(Cell_handle c, bool verbose = false) const*

Checks the combinatorial and geometric validity of the cell (see Section 41.2). Also checks that the circumscribing sphere of cells is empty.
When *verbose* is set to true, messages are printed to give a precise indication of the kind of invalidity encountered.

These methods are mainly a debugging help for the users of advanced features.

advanced

See Also

Periodic_3_triangulation_3

Class

CGAL::Periodic_3_triangulation_hierarchy_3<PTr>

Definition

The class *Periodic_3_triangulation_hierarchy_3* implements a triangulation augmented with a data structure which allows fast point location queries.

#include <CGAL/Periodic_3_triangulation_hierarchy_3.h>

Parameters

It is templated by a parameter which must be instantiated by one of the CGAL periodic triangulation classes. *In the current implementation, only Periodic_3_Delaunay_triangulation_3 is supported for PTr.*

PTr::Vertex has to be a model of the concept *Periodic_3TriangulationHierarchyVertexBase_3*.
PTr::Geom_traits has to be a model of the concept *Periodic_3DelaunayTriangulationTraits_3*.

Inherits From

PTr

Periodic_3_triangulation_hierarchy_3<PTr> offers exactly the same functionalities as *PTr*. Most of them (point location, insertion, removal. . .) are overloaded to improve their efficiency by using the hierarchic structure.

Note that, since the algorithms that are provided are randomized, the running time of constructing a triangulation with a hierarchy may be improved when shuffling the data points.

However, the I/O operations are not overloaded. So, writing a hierarchy into a file will lose the hierarchic structure and reading it from the file will result in an ordinary triangulation whose efficiency will be the same as *PTr*.

Implementation

The data structure is a hierarchy of triangulations. The triangulation at the lowest level is the original triangulation where operations and point location are to be performed. Then at each succeeding level, the data structure stores a triangulation of a small random sample of the vertices of the triangulation at the preceding level. Point location is done through a top-down nearest neighbor query. The nearest neighbor query is first performed naively in the top level triangulation. Then, at each following level, the nearest neighbor at that level is found through a linear walk performed from the nearest neighbor found at the preceding level. Because the number of vertices in each triangulation is only a small fraction of the number of vertices of the preceding triangulation the data structure remains small and achieves fast point location queries on real data.

See Also

CGAL::Periodic_3_triangulation_hierarchy_vertex_base_3
CGAL::Periodic_3_Delaunay_triangulation_3

Periodic_3DelaunayTriangulationTraits_3

Concept

Definition

The concept Periodic_3DelaunayTriangulationTraits_3 is the first template parameter of the classes *Periodic_3_Delaunay_triangulation_3* and *Periodic_3_triangulation_3*. It refines the concept *DelaunayTriangulationTraits_3* from the CGAL 3D Triangulation (Page **??**) package. It redefines the geometric objects, predicates and constructions to work with point-offset pairs. In most cases the offsets will be (0,0,0) and the predicates from *DelaunayTriangulationTraits_3* can be used directly. For efficiency reasons we maintain for each functor the version without offsets.

Refines

DelaunayTriangulationTraits_3

In addition to the requirements described for the traits class *DelaunayTriangulationTraits_3*, the geometric traits class of a Periodic Delaunay triangulation must fulfill the following requirements:

Types

Periodic_3DelaunayTriangulationTraits_3:: Point_3	The point type. It must be a model of *Kernel::Point_3*.
Periodic_3DelaunayTriangulationTraits_3:: Vector_3	The vector type. It must be a model of *Kernel::Vector_3*.
Periodic_3DelaunayTriangulationTraits_3:: Periodic_3_offset_3	The offset type. It must be a model of the concept *Periodic_3Offset_3*.
Periodic_3DelaunayTriangulationTraits_3:: Iso_cuboid_3	A type representing an axis-aligned cuboid. It must be a model of *Kernel::Iso_cuboid_3*.

The following three types represent geometric primitives in $\mathbb{R}^3$. They are required to provide functions converting primitives from $\mathbb{T}_c^3$ to $\mathbb{R}^3$, i.e. constructing representatives in $\mathbb{R}^3$.

Periodic_3DelaunayTriangulationTraits_3:: Segment_3	A segment type. It must be a model of *Kernel::Segment_3*.
Periodic_3DelaunayTriangulationTraits_3:: Triangle_3	A triangle type. It must be a model of *Kernel::Triangle_3*.
Periodic_3DelaunayTriangulationTraits_3:: Tetrahedron_3	A tetrahedron type. It must be a model of *Kernel::Tetrahedron_3*.

Periodic_3DelaunayTriangulationTraits_3:: Compare_xyz_3

A predicate object that must provide the function operators
Comparison_result operator()(Point_3 p, Point_3 q),
which returns *EQUAL* if the two points are equal and
Comparison_result operator()(Point_3 p, Point_3 q, Periodic_3_offset_3 o_p, Periodic_3_offset_3 o_q),
which returns *EQUAL* if the two point-offset pairs are equal. Otherwise it must return a consistent order for any two points chosen in a same line.
Precondition: *p*, *q* lie inside the domain.

Periodic_3DelaunayTriangulationTraits_3:: Orientation_3

A predicate object that must provide the function operators
Orientation operator()(Point_3 p, Point_3 q, Point_3 r, Point_3 s),
which returns *POSITIVE*, if *s* lies on the positive side of the oriented plane *h* defined by *p*, *q*, and *r*, returns *NEGATIVE* if *s* lies on the negative side of *h*, and returns *COPLANAR* if *s* lies on *h* and
Orientation operator()(Point_3 p, Point_3 q, Point_3 r, Point_3 s, Periodic_3_offset_3 o_p, Periodic_3_offset_3 o_q, Periodic_3_offset_3 o_r, Periodic_3_offset_3 o_s),
which returns *POSITIVE*, if the point-offset pair *(s,o_s)* lies on the positive side of the oriented plane *h* defined by *(p,o_p)*, *(q,o_q)*, and *(r,o_r)*, returns *NEGATIVE* if *(s,o_s)* lies on the negative side of *h*, and returns *COPLANAR* if *(s,o_s)* lies on *h*.
Precondition: *p*, *q*, *r*, *s* lie inside the domain.

Periodic_3DelaunayTriangulationTraits_3:: Side_of_oriented_sphere_3

A predicate object that must provide the function operators
Oriented_side operator()(Point_3 p, Point_3 q, Point_3 r, Point_3 s, Point_3 t),
which determines on which side of the oriented sphere circumscribing *p, q, r, s* the point *t* lies and
Oriented_side operator()(Point_3 p, Point_3 q, Point_3 r, Point_3 s, Point_3 t, Periodic_3_offset_3 o_p, Periodic_3_offset_3 o_q, Periodic_3_offset_3 o_r, Periodic_3_offset_3 o_s, Periodic_3_offset_3 o_t),
which determines on which side of the oriented sphere circumscribing *(p,o_p), (q,o_q), (r,o_r), (s,o_s)* the point-offset pair *(t,o_t)* lies.
Precondition: *p*, *q*, *r*, *s*, *t* lie inside the domain.

Periodic_3DelaunayTriangulationTraits_3:: Compare_distance_3

A predicate object that must provide the function operators
Comparison_result operator()(Point_3 p, Point_3 q, Point_3 r),
which compares the distance between *p* and *q* to the distance between *p* and *r* and
Comparison_result operator()(Point_3 p, Point_3 q, Point_3 r, Periodic_3_offset_3 o_p, Periodic_3_offset_3 o_q, Periodic_3_offset_3 o_r),
which compares the distance between *(p,o_p)* and *(q,o_q)* to the distance between *(p,o_p)* and *(r,o_r)*.
Precondition: *p*, *q*, *r* lie inside the domain.

In addition, only when vertex removal is used, the traits class must provide the following predicate objects:
Periodic_3DelaunayTriangulationTraits_3:: Coplanar_orientation_3

A predicate object that must provide the function operators
Orientation operator()(Point_3 p, Point_3 q, Point_3 r),
which returns *COLLINEAR*, if the points are collinear; otherwise it must return a consistent orientation for any three points chosen in a same plane and
Orientation operator()(Point_3 p, Point_3 q, Point_3 r Periodic_3_offset_3 o_p, Periodic_3_offset_3 o_q, Periodic_3_offset_3 o_r),
which returns *COLLINEAR*, if the point-offset pairs are collinear; otherwise it must return a consistent orientation for any three point-offset pairs chosen in a same plane.
Precondition: *p*, *q*, *r* lie inside the domain.

Periodic_3DelaunayTriangulationTraits_3:: Coplanar_side_of_bounded_circle_3

A predicate object that must provide the function operators
Bounded_side operator()(Point_3 p, Point_3 q, Point_3 r, Point_3 s),
which determines the bounded side of the circle defined by *p, q*, and *r* on which the point *s* lies and
Bounded_side operator()(Point_3 p, Point_3 q, Point_3 r, Point_3 s, Periodic_3_offset_3 o_p, Periodic_3_offset_3 o_q, Periodic_3_offset_3 o_r, Periodic_3_offset_3 o_s),
which determines the bounded side of the circle defined by *(p,o_p), (q,o_q)*, and *(r,o_r)* on which the point-offset pair *(s,o_s)* lies.
Precondition: *p,q,r*, and *s* are coplanar and *p,q*, and *r* are not collinear, *(p,o_p),(q,o_q),(r,o_r)*, and *(s,o_s)* are coplanar and *(p,o_p),(q,o_q)*, and *(r,o_r)* are not collinear, respectively, and *p*, *q*, *r*, *s*, *t* lie inside the domain.

In addition, only when *is_Gabriel* is used, the traits class must provide the following predicate object: *Periodic_3DelaunayTriangulationTraits_3:: Side_of_bounded_sphere_3*

A predicate object that must provide the function operators
Bounded_side operator()(Point_3 p, Point_3 q, Point_3 t),
which returns the position of the point *t* relative to the sphere that has *pq* as its diameter,
Bounded_side operator()(Point_3 p, Point_3 q, Point_3 t, Periodic_3_offset_3 o_p, Periodic_3_offset_3 o_q, Periodic_3_offset_3 o_t),
which returns the position of the point-offset pair *(t,o_t)* relative to the sphere that has *(p,o_p)(q,o_q)* as its diameter,
Bounded_side operator()(Point_3 p, Point_3 q, Point_3 r, Point_3 t),
which returns the position of the point *t* relative to the sphere passing through *p, q*, and *r* and whose center is in the plane defined by these three points,
Bounded_side operator()(Point_3 p, Point_3 q, Point_3 r, Point_3 t, Periodic_3_offset_3 o_p, Periodic_3_offset_3 o_q, Periodic_3_offset_3 o_r, Periodic_3_offset_3 o_q),
which returns the position of the point-offset pair *(t,o_t)* relative to the sphere passing through *(p,o_p), (q,o_q)*, and *(r,o_r)* and whose center is in the plane defined by these three point-offset pairs,
Bounded_side operator()(Point_3 p, Point_3 q, Point_3 r, Point_3 s, Point_3 t),
which returns the relative position of point *t* to the sphere defined by *p, q, r*, and *s*; the order of the points *p, q, r*, and *s* does not matter, and
Bounded_side operator()(Point_3 p, Point_3 q, Point_3 r, Point_3 s, Point_3 t, Periodic_3_offset_3 o_p, Periodic_3_offset_3 o_q, Periodic_3_offset_3 o_r, Periodic_3_offset_3 o_s, Periodic_3_offset_3 o_q),
which returns the relative position of the point-offset pair *(t,o_t)* to the sphere defined by *(p,o_p), (q,o_q), (r,o_r)*, and *(s,o_s)*; the order of the point-offset pairs *(p,o_p), (q,o_q), (r,o_r)*, and *(s,o_s)* does not matter.
Precondition: *p, q, r*, and *s* are not coplanar, *(p,o_p), (q,o_q), (r,o_r)*, and *(s,o_s)* are not coplanar, *p*, *q*, *r*, *s*, *t* lie inside the domain.

Note that the traits must provide exact constructions in order to guarantee exactness of the following construction functors.

Periodic_3DelaunayTriangulationTraits_3:: Construct_point_3

A constructor object that must provide the function operator
Point_3 operator()(Point_3 p, Periodic_3_offset_3 o_p),
which constructs a point from a point-offset pair.
Precondition: *p* lies inside the domain.

Periodic_3DelaunayTriangulationTraits_3:: Construct_segment_3

A constructor object that must provide the function operators
Segment_3 operator()(Point_3 p, Point_3 q),
which constructs a segment from two points and
Segment_3 operator()(Point_3 p, Point_3 q, Periodic_3_offset_3 o_p, Periodic_3_offset_3 o_q),
which constructs a segment from two point-offset pairs.
Precondition: *p*, *q* lie inside the domain.

Periodic_3DelaunayTriangulationTraits_3:: Construct_triangle_3

A constructor object that must provide the function operators
Triangle_3 operator()(Point_3 p, Point_3 q, Point_3 r),
which constructs a triangle from three points and
Triangle_3 operator()(Point_3 p, Point_3 q, Point_3 r, Periodic_3_offset_3 o_q, Periodic_3_offset_3 o_q, Periodic_3_offset_3 o_r),
which constructs a triangle from three point-offset pairs.
Precondition: *p*, *q*, *r* lie inside the domain.

Periodic_3DelaunayTriangulationTraits_3:: Construct_tetrahedron_3

A constructor object that must provide the function operators
Tetrahedron_3 operator()(Point_3 p, Point_3 q, Point_3 r, Point_3 s),
which constructs a tetrahedron from four points and
Tetrahedron_3 operator()(Point_3 p, Point_3 q, Point_3 r, Point_3 s, Periodic_3_offset_3 o_q, Periodic_3_offset_3 o_q, Periodic_3_offset_3 o_r, Periodic_3_offset_3 o_s),
which constructs a tetrahedron from four point-offset pairs.
Precondition: *p*, *q*, *r*, *s* lie inside the domain.

In addition, only when the dual operations are used, the traits class must provide the following constructor object:

Periodic_3DelaunayTriangulationTraits_3:: Construct_circumcenter_3

A constructor object that must provide the function operators
Point_3 operator()(Point_3 p, Point_3 q, Point_3 r, Point_3 s),
which constructs the circumcenter of four points and
Point_3 operator()(Point_3 p, Point_3 q, Point_3 r, Point_3 s, Periodic_3_offset_3 o_p, Periodic_3_offset_3 o_q, Periodic_3_offset_3 o_r, Periodic_3_offset_3 o_s),
which constructs the circumcenter of four point-offset pairs.
Precondition: *p*, *q*, *r* and *s* as well as *(p,o_p)*, *(q,o_q)*, *(r,o_r)* and *(s,o_s)* must be non coplanar. *p*, *q*, *r*, *s* lie inside the domain.

The optional types must be provided in any case, however they can be replaced by dummy types if the respective functions are not used.

Creation

Periodic_3DelaunayTriangulationTraits_3 traits; Default constructor.

Periodic_3DelaunayTriangulationTraits_3 traits(Periodic_triangulation_traits_3 tr);

Copy constructor.

Access Functions

void *traits.set_domain(Iso_cuboid_3 domain)*

Set the size of the fundamental domain. This is necessary to evaluate predicates correctly.
Precondition: *domain* represents a cube.

Operations

The following functions give access to the predicate and construction objects:

Compare_xyz_3	*traits.compare_xyz_3_object()*
Orientation_3	*traits.orientation_3_object()*

Side_of_oriented_sphere_3	*traits.side_of_oriented_sphere_3_object()*
Compare_distance_3	*traits.compare_distance_3_object()*

The following functions must be provided if vertex removal is used; otherwise dummy functions can be provided.

Coplanar_orientation_3	*traits.coplanar_3_orientation_3_object()*
Coplanar_side_of_bounded_circle_3	*traits.coplanar_side_of_bounded_circle_3_object()*

The following function must be provided only if the *is_Gabriel* methods of *Periodic_3_Delaunay_triangulation_3* are used; otherwise a dummy function can be provided.

Side_of_bounded_sphere_3	*traits.side_of_bounded_sphere_3_object()*

Construct_segment_3	*traits.construct_segment_3_object()*
Construct_triangle_3	*traits.construct_triangle_3_object()*
Construct_tetrahedron_3	*traits.construct_tetrahedron_3_object()*

The following function must be provided only if the methods of *Periodic_3_Delaunay_triangulation_3* returning elements of the Voronoi diagram are used; otherwise a dummy function can be provided:

Construct_circumcenter_3	*traits.construct_circumcenter_3_object()*

Has Models

CGAL::Periodic_3_triangulation_traits_3

See Also

DelaunayTriangulationTraits_3

CGAL::Periodic_3_triangulation_traits_3<Traits,Periodic_3Offset_3>

Definition

The class *Periodic_3_triangulation_traits_3<Traits,Periodic_3Offset_3>* is designed as a default traits class for the class *Periodic_3_triangulation_3<Periodic_3TriangulationTraits_3,TriangulationDataStructure_3>*.

The argument *Traits* must be a model of the *DelaunayTriangulationTraits_3* concept. The argument *Periodic_3Offset_3* must be a model of the concept *Periodic_3Offset_3* and defaults to *Periodic_3_offset_3*.

Note that this template class is specialized for *CGAL::Filtered_kernel*, so that it automatically provides filtered predicates. This holds implicitly for *CGAL::Exact_predicates_inexact_constructions_kernel*, as it is an instantiation of *CGAL::Filtered_kernel*.

#include <CGAL/Periodic_3_triangulation_traits_3.h>

Is Model for the Concepts

Periodic_3DelaunayTriangulationTraits_3

Inherits From

Traits

Concept

Periodic_3TriangulationDSCellBase_3

Definition

At the base level (see Sections 41.5 and 40.2), a cell stores handles to its four vertices and to its four neighbor cells. The vertices and neighbors are indexed 0, 1, 2 and 3. Neighbor i lies opposite to vertex i.

Refines

TriangulationDSCellBase_3

For periodic triangulation the cell base class needs to additionally store an offset for each vertex. Only the last three bits of each integer are required to be stored. The remaining part does not contain any information.

Access Functions

int *c.offset(int i) const*

Returns the offset of vertex i.
Precondition: $i \in \{0, 1, 2, 3\}$.

Setting

void *c.set_offsets(int off0, int off1, int off2, int off3)*

Sets the vertex offsets according to *off0* to *off3*.

Has Models

CGAL::Periodic_3_triangulation_ds_cell_base_3

See Also

TriangulationDataStructure_3
TriangulationDSCellBase_3
Periodic_3TriangulationDSVertexBase_3

Concept

Periodic_3TriangulationDSVertexBase_3

Definition

At the base level of 3D-triangulations (see Sections 41.5 and 40.2), a vertex provides access to one of its incident cells through a handle.

For optimization purposes we need to extend the concept *TriangulationDSVertexBase_3* as follows.

Refines

TriangulationDSVertexBase_3

Periodic_3TriangulationDSVertexBase_3 needs to store an offset and some index inside the vertex.

Types

Periodic_3TriangulationDSVertexBase_3:: Periodic_3_offset_3	A model of the concept *Periodic_3Offset_3*

Access Functions

Periodic_3_offset_3	*v.offset() const*	Returns the offset stored in the vertex.
bool	*v.get_offset_flag() const*	Returns *true* if the offset has been set, *false* otherwise.
void	*v.set_offset(Periodic_3_offset_3 o)*	Sets the offset and sets the offset flag to *true*.
void	*v.clear_offset()*	Sets the offset flag to *false* and clears the offset.

Has Models

CGAL::Periodic_3_triangulation_ds_vertex_base_3

See Also

TriangulationDataStructure_3
TriangulationDSVertexBase_3
Periodic_3TriangulationDSCellBase_3

CGAL::Periodic_3_triangulation_ds_cell_base_3<>

Class

Definition

The class *Periodic_3_triangulation_ds_cell_base_3* is a model of the concept *Periodic_3TriangulationDSCellBase_3* to be used by *Triangulation_data_structure_3* to represent cells of a periodic triangulation.

#include <CGAL/Periodic_3_triangulation_ds_cell_base_3.h>

Is Model for the Concepts

Periodic_3TriangulationDSCellBase_3

See Also

CGAL::Periodic_3_triangulation_ds_vertex_base_3
CGAL::Triangulation_cell_base_3
CGAL::Triangulation_cell_base_with_info_3
CGAL::Triangulation_cell_base_with_circumcenter_3

Class

CGAL::Periodic_3_triangulation_ds_vertex_base_3<>

Definition

The class *Periodic_3_triangulation_ds_vertex_base_3* is a model of the concept *Periodic_3TriangulationDSVertexBase_3* to be used by *Triangulation_data_structure_3* to represent vertices of a periodic triangulation.

#include <CGAL/Periodic_3_triangulation_ds_vertex_base_3.h>

Is Model for the Concepts

Periodic_3TriangulationDSVertexBase_3

See Also

CGAL::Periodic_3_triangulation_ds_cell_base_3
CGAL::Triangulation_vertex_base_3
CGAL::Triangulation_vertex_base_with_info_3

Periodic_3Offset_3

Definition

The concept Periodic_3Offset_3 describes a three-dimensional integer vector with some specialized access functions and operations.

Creation

Periodic_3Offset_3 o;	Default constructor.
Periodic_3Offset_3 o(int x, int y, int z);	Constructs the offset (x,y,z).

Operations

Periodic_3Offset_3	$o + o'$	Return the vector sum of *o* and *o'*.
Periodic_3Offset_3	$o - o'$	Return the vector difference of *o* and *o'*.
Periodic_3Offset_3	$-o$	Return the negative vector of *o*.
void	$o += o'$	Add *o'* to *o* using vector addition.
void	$o -= o'$	Subtract *o'* from *o* using vector subtraction.
bool	$o == o'$	Return *true* if *o'* and *o* represent the same vector.
bool	$o \mathrel{!=} o'$	Return *true* if *o'* and *o* do not represent the same vector.
bool	$o < o'$	Compare *o* and *o'* lexicographically.

Access Functions

int	*o*[*int i*]	Return the *i*-th entry of *o*.
int	*o.x() const*	Return the *x*-entry of *o*.
int	*o.y() const*	Return the *y*-entry of *o*.
int	*o.z()*	Return the *z*-entry of *o*.
bool	*o.is_null() const*	Returns *true* if *o* is equal to (0,0,0).

I/O

istream&	*istream & is >> Periodic_3_offset_3 & off*	
		Inputs an offset from *is*.
ostream&	*ostream & os << Periodic_3_offset_3 & off*	
		Outputs an offset from *os*.

Has Models

CGAL::Periodic_3_offset_3

See Also

Periodic_3DelaunayTriangulationTraits_3

CGAL::Periodic_3_offset_3

Class

Definition

The class *Periodic_3_offset_3* is a model of the concept *Periodic_3Offset_3*.

#include <CGAL/Periodic_3_offset_3.h>

Is Model for the Concepts

Periodic_3Offset_3

Enum

CGAL::Periodic_3_triangulation_3::Iterator_type

Definition

The enum *Iterator_type* is defined by *Periodic_3_triangulation_3* to specify the behavior of geometric iterators.

enum Iterator_type { *STORED=0, UNIQUE, STORED_COVER_DOMAIN, UNIQUE_COVER_DOMAIN*}*;*

The elements of the enum have the following meaning:

- STORED: Return all geometric primitives as they are stored internally in *Triangulation_data_structure_3*.
- UNIQUE: Return only one representative of each geometric primitive even if the triangulation is computed in a multiply sheeted covering space. Choose the representative whose maximum offset is minimal but non-negative in each direction of space.
- STORED_COVER_DOMAIN: Same as STORED but return additionally all primitives whose intersection with the original domain of the current covering space is non-empty.
- UNIQUE_COVER_DOMAIN: Same as UNIQUE but return additionally all primitives whose intersection with the original domain is non-empty.

See Also

CGAL::Periodic_3_triangulation_3

CGAL::Periodic_3_triangulation_3::Locate_type

Definition

The enum *Locate_type* is defined by *Periodic_3_triangulation_3* to specify which case occurs when locating a point in the triangulation. If the triangulation does not contain any points *EMPTY* is returned.

enum Locate_type { *VERTEX=0, EDGE, FACET, CELL, EMPTY*}*;*

See Also

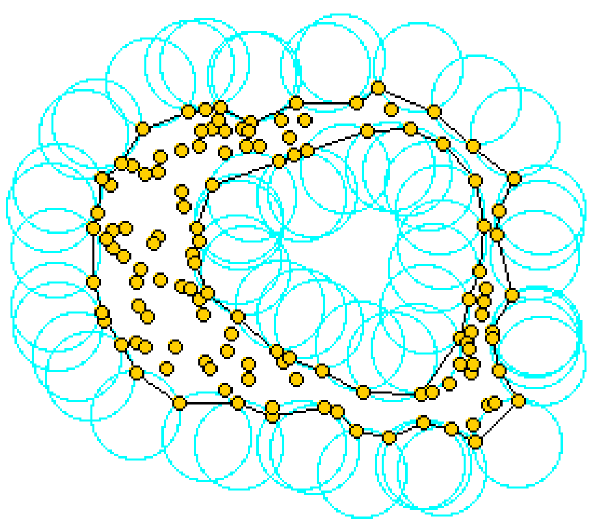
CGAL::Periodic_3_triangulation_3

Chapter 42

2D Alpha Shapes

Tran Kai Frank Da

Contents



Assume we are given a set S of points in 2D or 3D and we'd like to have something like "the shape formed by these points." This is quite a vague notion and there are probably many possible interpretations, the α-shape being one of them. Alpha shapes can be used for shape reconstruction from a dense unorganized set of data points. Indeed, an α-shape is demarcated by a frontier, which is a linear approximation of the original shape [BB97].

As mentioned in Edelsbrunner's and Mücke's paper [EM94], one can intuitively think of an α-shape as the following. Imagine a huge mass of ice-cream making up the space $\mathbb{R}^3$ and containing the points as "hard" chocolate pieces. Using one of these sphere-formed ice-cream spoons we carve out all parts of the ice-cream block we can reach without bumping into chocolate pieces, thereby even carving out holes in the inside (e.g. parts not reachable by simply moving the spoon from the outside). We will eventually end up with a (not necessarily convex) object bounded by caps, arcs and points. If we now straighten all "round" faces to triangles and line segments, we have an intuitive description of what is called the α-shape of S. Here's an example for this process in 2D (where our ice-cream spoon is simply a circle):

And what is α in the game? α is the squared radius of the carving spoon. A very small value will allow us to eat up all of the ice-cream except the chocolate points themselves. Thus we already see that the α-shape degenerates to the point-set S for $\alpha \rightarrow 0$. On the other hand, a huge value of α will prevent us even from moving the spoon between two points since it's way too large. So we will never spoon up ice-cream lying in the inside of the convex hull of S, and hence the α-shape for $\alpha \rightarrow \infty$ is the convex hull of S.[1]

42.1 Definitions

We distinguish two versions of alpha shapes. *Basic alpha shapes* are based on the Delaunay triangulation. *Weighted alpha shapes* are based on its generalization, the regular triangulation, replacing the euclidean distance by the power to weighted points.

There is a close connection between alpha shapes and the underlying triangulations. More precisely, the α-complex of S is a subcomplex of this triangulation of S, containing the α-exposed k-simplices, $0 \leq k \leq d$. A simplex is α-exposed, if there is an open disk (resp. ball) of radius $\sqrt{\alpha}$ through the vertices of the simplex that does not contain any other point of S, for the metric used in the computation of the underlying triangulation. The corresponding α-shape is defined as the underlying interior space of the α-complex (see [EM94]).

In general, an α-complex is a non-connected and non-pure polytope, it means, that one k-simplex, $0 \leq k \leq d-1$ is not necessary adjacent to a $(k+1)$-simplex.

The α-shapes of S form a discrete family, even though they are defined for all real numbers α with $0 \leq \alpha \leq \infty$. Thus, we can represent the entire family of α-shapes of S by the underlying triangulation of S. In this representation each k-simplex of the underlying triangulation is associated with an interval that specifies for which values of α the k-simplex belongs to the α-shape. Relying on this fact, the family of α-shapes can be computed efficiently and relatively easily. Furthermore, we can select an appropriate α-shape from a finite number of different α-shapes and corresponding α-values.

42.2 Functionality

The class *CGAL::Alpha_shape_2<Dt>* represents the family of α-shapes of points in a plane for *all* positive α. It maintains the underlying triangulation *Dt* which represents connectivity and order among squared radius of its faces. Each k-dimensional face of the *Dt* is associated with an interval that specifies for which values of

[1] ice cream, ice cream!!! The wording of this introductory paragraphs is borrowed from Kaspar Fischer's " Introduction to Alpha Shapes" which can be found at http://people.inf.ethz.ch/fischerk/pubs/as.pdf. The picture has been taken from Walter Luh's homepage at http://www.stanford.edu/w̃luh/cs448b/alphashapes.html.

α the face belongs to the α-shape. There are links between the intervals and the *k*-dimensional faces of the triangulation.

The class *CGAL::Alpha_shape_2<Dt>* provides functions to set and get the current α-value, as well as an iterator that enumerates the α-values where the α-shape changes.

It provides iterators to enumerate the vertices and edges that are in the α-shape, and functions that allow to classify vertices, edges and faces with respect to the α-shape. They can be in the interior of a face that belongs or does not belong to the α-shape. They can be singular/regular, that is be on the boundary of the α-shape, but not incident/incident to a triangle of the α-complex.

Finally, it provides a function to determine the α-value such that the α-shape satisfies the following two properties, or at least the second one if there is no such α that both are satisfied:

(i) The number of components equals a number of your choice and
(ii) all data points are either on the boundary or in the interior of the regularized version of the α-shape (no singular edges).

The current implementation is static, that is after its construction points cannot be inserted or removed.

42.3 Concepts and Models

We currently do not specify concepts for the underlying triangulation type. Models that work for a basic alpha-shape are the classes *CGAL::Delaunay_triangulation_2* and *CGAL::Triangulation_hierarchy_2* templated with a Delaunay triangulation. A model that works for a weighted alpha-shape is the class *CGAL::Regular_triangulation_2*.

The triangulation needs a geometric traits class as argument. The requirements of this class are described in the concept *CGAL::AlphaShapeTraits_2* for which the CGAL kernels and *CGAL::Weighted_alpha_shape_euclidean_traits_2* are models.

There are no requirements on the triangulation data structure. However it must be parameterized with vertex and face classes, which are model of the concepts *AlphaShapeVertex_2* and *AlphaShapeFace_2*, by default the classes *CGAL::Alpha_shape_vertex_base_2<Gt>* and *CGAL::Alpha_shape_face_base_2<Gt>*.

42.4 Examples

42.4.1 Example for Basic Alpha-Shapes

The basic alpha shape needs a Delaunay triangulation as underlying triangulation *Dt*. The Delaunay triangulation class is parameterized with a geometric and a triangulation data structure traits.

For the geometric traits class we can use a CGAL kernel.

For the triangulation data structure traits, we have to choose the vertex and face classes needed for alpha shapes, namely *CGAL::Alpha_shape_vertex_base_2<Gt, Dv>* and *CGAL::Alpha_shape_face_base_2<Gt,Df>*. As default vertex and face type they use *CGAL::Triangulation_vertex_base_2<Gt>* and *CGAL::Triangulation_face_base_2<Gt>* respectively.

The following code snippet shows how to obtain a basic alpha shape type.

```
typedef CGAL::Exact_predicates_inexact_constructions_kernel K;

typedef CGAL::Alpha_shape_vertex_base_2<K> Av;

typedef CGAL::Triangulation_face_base_2<K> Tf;
typedef CGAL::Alpha_shape_face_base_2<K,Tf> Af;

typedef CGAL::Triangulation_default_data_structure_2<K,Av,Af> Tds;
typedef CGAL::Delaunay_triangulation_2<K,Tds> Dt;
typedef CGAL::Alpha_shape_2<Dt> Alpha_shape_2;
```

42.4.2 Example for Basic Alpha-Shapes with Many Points

When the input data set is huge, say more than 10.000 points, it pays off to use a triangulation hierarchy. It has the same API as the Delaunay triangulation and differs only in the types of the vertices and faces. Therefore, the only part that changes are the typedefs in the beginning.

```
typedef CGAL::Exact_predicates_inexact_constructions_kernel K;

typedef CGAL::Alpha_shape_vertex_base_2<K> Avb;
typedef CGAL::Triangulation_hierarchy_vertex_base_2<Avb> Av

typedef CGAL::Triangulation_face_base_2<K> Tf;
typedef CGAL::Alpha_shape_face_base_2<K,Tf> Af;

typedef CGAL::Triangulation_default_data_structure_2<K,Av,Af> Tds;
typedef CGAL::Delaunay_triangulation_2<K,Tds> Dt;
typedef CGAL::Triangulation_hierarchy_2<Dt> Ht;
typedef CGAL::Alpha_shape_2<Ht> Alpha_shape_2;
```

42.4.3 Example for Weighted Alpha-Shapes

A weighted alpha shape, needs a regular triangulation as underlying triangulation *Dt*, and it needs a particular face class, namely *CGAL::Regular_triangulation_face_base_2<Gt>*. Note that there is no special weighted alpha shape class.

```
typedef CGAL::Exact_predicates_inexact_constructions_kernel K;
typedef CGAL::Weighted_alpha_shape_euclidean_traits_2<K> Gt;

typedef CGAL::Alpha_shape_vertex_base_2<Gt> Av;

typedef CGAL::Regular_triangulation_face_base_2<Gt> Rf;
typedef CGAL::Alpha_shape_face_base_2<Gt,Rf>  Af;

typedef CGAL::Triangulation_default_data_structure_2<Gt,Av,Af> Tds;
typedef CGAL::Regular_triangulation_2<Gt,Tds> Rt;
typedef CGAL::Alpha_shape_2<Rt> Alpha_shape_2;
```

2D Alpha Shapes Reference Manual

Tran Kai Frank Da

This chapter presents a framework for alpha shapes. The description is based on the articles [EM94, Ede92]. Alpha shapes are the generalization of the convex hull of a point set. Let S be a finite set of points in $\mathbb{R}^d$, $d = 2,3$ and α a parameter with $0 \leq \alpha \leq \infty$. For $\alpha = \infty$, the α-shape is the convex hull of S. As α decreases, the α-shape shrinks and develops cavities, as soon as a sphere of radius $\sqrt{\alpha}$ can be put inside. Finally, for $\alpha = 0$, the α-shape is the set S itself.

We distinguish two versions of alpha shapes, one is based on the Delaunay triangulation and the other on its generalization, the regular triangulation, replacing the natural distance by the power to weighted points. The metric used determines an underlying triangulation of the alpha shape and thus, the version computed. The *basic alpha shape* (cf. 42.4.1) is associated with the Delaunay triangulation (cf. 37.5). The *weighted alpha shape* (cf. 42.4.3) is associated with the regular triangulation (cf. 37.6).

There is a close connection between alpha shapes and the underlying triangulations. More precisely, the α-complex of S is a subcomplex of this triangulation of S, containing the α-exposed k-simplices, $0 \leq k \leq d$. A simplex is α-exposed, if there is an open disk (resp. ball) of radius $\sqrt{\alpha}$ through the vertices of the simplex that does not contain any other point of S, for the metric used in the computation of the underlying triangulation. The corresponding α-shape is defined as the underlying interior space of the α-complex.

In general, an α-complex is a non-connected and non-pure polytope, it means, that one k-simplex, $0 \leq k \leq d-1$ is not necessary adjacent to a $(k+1)$-simplex.

The α-shapes of S form a discrete family, even though they are defined for all real numbers α with $0 \leq \alpha \leq \infty$. Thus, we can represent the entire family of α-shapes of S by the underlying triangulation of S. In this representation each k-simplex of the underlying triangulation is associated with an interval that specifies for which values of α the k-simplex belongs to the α-shape. Relying on this result, the family of α-shapes can be computed efficiently and relatively easily. Furthermore, we can select an appropriate α-shape from a finite number of different α-shapes and corresponding α-values.

42.5 Classified Reference Pages

Concepts

Classes

42.6 Alphabetical List of Reference Pages

CGAL::Alpha_shape_2<Dt,ExactAlphaComparisonTag>

Class

Definition

The class *Alpha_shape_2<Dt,ExactAlphaComparisonTag>* represents the family of α-shapes of points in a plane for *all* positive α. It maintains the underlying triangulation *Dt* which represents connectivity and order among its faces. Each k-dimensional face of the *Dt* is associated with an interval that specifies for which values of α the face belongs to the α-shape. There are links between the intervals and the k-dimensional faces of the triangulation.

Note that this class is at the same time used for *basic* and for *weighted* Alpha Shapes.

#include <CGAL/Alpha_shape_2.h>

Parameters

The template parameter *Dt* has to be either *Delaunay_triangulation_2* or *Regular_triangulation_2*. Note that *DT::Geom_traits*, *DT::Vertex* and *DT::Face* must model the concepts *AlphaShapeTraits_2*, *AlphaShapeVertex_2* and *AlphaShapeFace_2* respectively.

The template parameter *ExactAlphaComparisonTag* is a tag that, when set to *CGAL::Tag_true*, triggers exact comparisons between alpha values. This is useful when the underlying triangulation is instantiated with an exact predicates inexact constructions kernel. By default the *ExactAlphaComparisonTag* is set to *CGAL::Tag_false* as it induces a small overhead. Note that since such a strategy does not make sense if used together with a traits class with exact constructions, the tag *ExactAlphaComparisonTag* is not taken into account if *Dt::Geom_traits::FT* is not a floating point number type.

Inherits From

Dt

This class is the underlying triangulation class.

The modifying functions *insert* and *remove* will overwrite the inherited functions. At the moment, only the static version is implemented.

Types

Alpha_shape_2<Dt,ExactAlphaComparisonTag>:: Gt

the alpha shape traits type.

it has to derive from a triangulation traits class. For example *Dt::Point* is a Point class.

typedef Gt::FT

FT; the number type for computation.

Alpha_shape_2<Dt,ExactAlphaComparisonTag>:: size_type

The size type.

Alpha_shape_2<Dt,ExactAlphaComparisonTag>:: Alpha_iterator

A bidirectional and non-mutable iterator that allow to traverse the increasing sequence of different α-values.
Precondition: Its *value_type* is *FT*

Alpha_shape_2<Dt,ExactAlphaComparisonTag>:: Alpha_shape_vertices_iterator

A bidirectional and non-mutable iterator that allow to traverse the vertices which belongs to the α-shape for the current α.
Precondition: Its *value_type* is *Dt::Vertex_handle*

Alpha_shape_2<Dt,ExactAlphaComparisonTag>:: Alpha_shape_edges_iterator

A bidirectional and non-mutable iterator that allow to traverse the edges which belongs to the α-shape for the current α.
Precondition: Its *value_type* is *Dt::Edge*.

enum Classification_type { *EXTERIOR, SINGULAR, REGULAR, INTERIOR*};

Distinguishes the different cases for classifying a k-dimensional face of the underlying triangulation of the α-shape.
EXTERIOR if the face does not belong to the α-complex.
SINGULAR if the face belongs to the boundary of the α-shape, but is not incident to any 2-dimensional face of the α-complex
REGULAR if the face belongs to the boundary of the α-shape and is incident to a 2-dimensional face of the α-complex
INTERIOR if the face belongs to the α-complex, but does not belong to the boundary of the α-shape.

enum Mode { *GENERAL, REGULARIZED*};

In general, an alpha shape can be disconnected and contain many singular edges or vertices. Its regularized version is formed by the set of regular edges and their vertices.

Creation

Alpha_shape_2<Dt,ExactAlphaComparisonTag> A(FT alpha = 0, Mode m = GENERAL);

Introduces an empty α-shape *A* for a positive α-value *alpha*.
Precondition: $alpha \geq 0$.

Alpha_shape_2<Dt,ExactAlphaComparisonTag> A(Dt& dt, FT alpha = 0, Mode m = GENERAL);

Builds an alpha shape of mode *m* from the triangulation *dt* for a positive α-value *alpha*. Be careful that this operation destroys the triangulation.
Precondition: *alpha* $\geq$ 0.

template < class InputIterator >
Alpha_shape_2<Dt,ExactAlphaComparisonTag> A(InputIterator first, InputIterator last, FT alpha = 0, Mode m = GENERAL)

Initializes the family of alpha-shapes with the points in the range [*first*, *last*) and introduces an α-shape *A* for a positive α-value *alpha*.
Precondition: The *value_type* of *first* and *last* is *Point*.
alpha $\geq$ 0.

Operations

template < class InputIterator >
std::ptrdiff_t A.make_alpha_shape(InputIterator first, InputIterator last)

Initialize the family of alpha-shapes with the points in the range [*first*, *last*). Returns the number of inserted points.
If the function is applied to an non-empty family of alpha-shape, it is cleared before initialization.
Precondition: The *value_type* of *first* and *last* is *Point*.

void *A.clear()* Clears the structure.

FT *A.set_alpha(FT alpha)*

Sets the α-value to *alpha*. Returns the previous α-value.
Precondition: *alpha* $\geq$ 0.

FT *A.get_alpha(void) const*

Returns the current α-value.

FT *A.get_nth_alpha(size_type n) const*

Returns the *n*-th alpha-value, sorted in an increasing order.
Precondition: *n* < number of alphas.

size_type *A.number_of_alphas() const*

Returns the number of different alpha-values.

Mode *A.set_mode(Mode m = GENERAL)*

Sets A to its general or regularized version. Returns the previous mode.

Mode *A.get_mode(void) const*

Returns whether A is general or regularized.

Alpha_shape_vertices_iterator

A.alpha_shape_vertices_begin()

Starts at an arbitrary finite vertex which belongs to the α-shape for the current α.

Alpha_shape_vertices_iterator

A.alpha_shape_vertices_end()

Past-the-end iterator.

Alpha_shape_edges_iterator

A.alpha_shape_edges_begin()

Starts at an arbitrary finite edge which belongs to the α-shape for the current α. In regularized mode, edges are represented as a pair (f,i), where f is an interior face of the α-shape.

Alpha_shape_edges_iterator

A.alpha_shape_edges_end()

Past-the-end iterator.

Predicates

Classification_type

A.classify(Point p, FT alpha = get_alpha())

Locates a point p in the underlying triangulation and Classifies the associated k-face with respect to A.

Classification_type

A.classify(Face_handle f, FT alpha = get_alpha())

Classifies the face f of the underlying triangulation with respect to A.

Classification_type

A.classify(Edge e, FT alpha = get_alpha())

Classifies the edge *e* of the underlying triangulation with respect to *A*.

Classification_type

A.classify(Face_handle f, int i, FT alpha = get_alpha())

Classifies the edge of the face *f* opposite to the vertex with index *i* of the underlying triangulation with respect to *A*.

Classification_type

A.classify(Vertex_handle v, FT alpha = get_alpha())

Classifies the vertex *v* of the underlying triangulation with respect to *A*.

Traversal of the α-Values

Alpha_iterator

A.alpha_begin() const

Returns an iterator that allows to traverse the sorted sequence of α-values of the family of alpha shapes.

Alpha_iterator

A.alpha_end() const

Returns the corresponding past-the-end iterator.

Alpha_iterator

A.alpha_find(FT alpha) const

Returns an iterator pointing to an element with α-value *alpha*, or the corresponding past-the-end iterator if such an element is not found.

Alpha_iterator

A.alpha_lower_bound(FT alpha) const

Returns an iterator pointing to the first element with α-value not less than *alpha*.

Alpha_iterator

A.alpha_upper_bound(FT alpha) const

Returns an iterator pointing to the first element with α-value greater than *alpha*.

Operations

size_type *A.number_of_solid_components(FT alpha = get_alpha())*

Returns the number of solid components of *A*, that is, the number of components of its regularized version.

Alpha_iterator

A.find_optimal_alpha(size_type nb_components) const

Returns an iterator pointing to the first element with α-value such that *A* satisfies the following two properties:
nb_components equals the number of solid components and
all data points are either on the boundary or in the interior of the regularized version of *A*.
If no such value is found, the iterator points to the first element with α-value such that *A* satisfies the second property.

I/O

The I/O operators are defined for *iostream*. The format for the iostream is an internal format.

#include <CGAL/IO/io.h>

ostream& *ostream& os << Alpha_shape_2<Dt> A*

Inserts the alpha shape *A* for the current α-value into the stream *os*.
Precondition: The insert operator must be defined for *Point*.

Implementation

The set of intervals associated with the *k*-dimensional faces of the underlying triangulation are stored in *multimaps*.

The cross links between the intervals and the *k*-dimensional faces of the triangulation are realized using methods in the *k*-dimensional faces themselves.

A.alpha_find uses linear search, while *A.alpha_lower_bound* and *A.alpha_upper_bound* use binary search. *A.number_of_solid_components* performs a graph traversal and takes time linear in the number of faces of the underlying triangulation. *A.find_optimal_alpha* uses binary search and takes time $O(n \log n)$, where n is the number of points.

AlphaShapeFace_2

Concept

Definition

Refines

TriangulationFaceBase_2.

Types

AlphaShapeFace_2:: Interval_3	A container type to get (and put) the three special values $(\alpha_1, \alpha_2, \alpha_3)$ associated with an alpha shape edge.
AlphaShapeFace_2:: FT	A type to hold a coordinate type class. The type must provide a copy constructor, assignment, comparison operators, negation, multiplication, division and allow the declaration and initialization with a small integer constant (cf. requirements for number types). An obvious choice would be coordinate type of the point class

Creation

advanced

AlphaShapeFace_2 f; — default constructor.

AlphaShapeFace_2 f(Vertex_handle v0, Vertex_handle v1, Vertex_handle v2);

constructor setting the incident vertices.

AlphaShapeFace_2 f(Vertex_handle v0,
Vertex_handle v1,
Vertex_handle v2,
Face_handle n0,
Face_handle n1,
Face_handle n2)

constructor setting the incident vertices and the neighboring faces.

advanced

Access Functions

Interval_3	*f.get_ranges(int i)*	returns the interval associated with the edge indexed with *i*, which contains three alpha values $\alpha_1 \leq \alpha_2 \leq \alpha_3$, such as for α between α_1 and α_2, the edge indexed with *i* is attached but singular, for α between α_2 and α_3, the edge is regular, and for α greater than α_3, the edge is interior.

FT	*f.get_alpha()*	return the alpha value, under which the alpha shape contains the face.

Modifiers

advanced

void	*f.set_ranges(int i, Interval_3 V)*	
		sets the interval associated with the edge indexed with *i*, which contains three alpha values $\alpha_1 \leq \alpha_2 \leq \alpha_3$, such as for α between α_1 and α_2, the edge indexed with *i* is attached but singular, for α between α_2 and α_3, the edge is regular, and for α greater than α_3, the edge is interior.
void	*f.set_alpha(FT A)*	sets the alpha value, under which the alpha shape contains the face.

advanced

CGAL::Alpha_shape_face_base_2<Traits, Fb, ExactAlphaComparisonTag>

Class

Definition

The class *Alpha_shape_face_base_2<Traits, Fb, ExactAlphaComparisonTag>* is the default model for the concept *AlphaShapeFace_2*.

#include <CGAL/Alpha_shape_face_base_2.h>

Parameters

The template parameter *Traits* has to be a model of *AlphaShapeTraits_2*.

The template parameter *Fb* has to be a model of *TriangulationFaceBase_2* (or *RegularTriangulationFaceBase_2*) if *Alpha_shape_face_base_2<Traits, Fb, ExactAlphaComparisonTag>* is intended to be used with an alpha-shape class based on a *Delaunay_triangulation_2* (or a *Regular_triangulation_2*).

The template parameter *ExactAlphaComparisonTag* is a tag that, when set to *CGAL::Tag_true*, triggers exact comparisons between alpha values. See the description provided in the documentation of *Alpha_shape_2* for more details. The default value is *CGAL::Tag_false*.

Is Model for the Concepts

AlphaShapeFace_2

Inherits From

Fb

Concept

AlphaShapeTraits_2

Definition

A model of the concept AlphaShapeTraits_2 must provide the following predicate and operations in addition to the requirements for the underlying triangulation traits class. It means, the metric has to be Euclidean for Delaunay triangulation or the power metric for regular triangulation.

Refines

TriangulationTraits_2

Has Models

The kernels supplied by CGAL are models of AlphaShapeTraits_2.

Projection_traits_xy_3<K> . . . . page **??**

Types

AlphaShapeTraits_2:: FT

A type to hold a coordinate type class. The type must provide a copy constructor, assignment, comparison operators, negation, multiplication, division and allow the declaration and initialization with a small integer constant (cf. requirements for number types).
Precondition: An obvious choice would be coordinate type of the point class.

Creation

Only a default constructor is required. Note that further constructors can be provided.

AlphaShapeTraits_2 t;

A default constructor.

Constructions by function objects

Compute_squared_radius_2

t.compute_squared_radius_2_object()

Returns an object, which has to be able to compute the squared radius of the circle of the points *p0, p1, p2* or the squared radius of smallest circle of the points *p0, p1*, as *FT* associated with *the metric used by Dt*.

Predicate by function object

Side_of_bounded_circle_2

t.side_of_bounded_circle_2_object()

Returns an object, which has to be able to compute the relative position of point *test* to the smallest circle of the points *p0, p1*, using *the same metric as Dt*.

Class

CGAL::Weighted_alpha_shape_euclidean_traits_2<K>

Definition

The class *Weighted_alpha_shape_euclidean_traits_2<K>* is the default model for the concept *AlphaShapeTraits_2* for the regular version of Alpha Shapes. *K* must be a kernel.

#include <CGAL/Weighted_alpha_shape_euclidean_traits_2.h>

Refines

Regular_triangulation_euclidean_traits_2<K, typename K::FT>

Is Model for the Concepts

AlphaShapeTraits_2

AlphaShapeVertex_2

Concept

Definition

Refines

TriangulationVertexBase_2.

Types

AlphaShapeVertex_2:: FT	A type to hold a coordinate type class. The type must provide a copy constructor, assignment, comparison operators, negation, multiplication, division and allow the declaration and initialization with a small integer constant (cf. requirements for number types). An obvious choice would be coordinate type of the point class.

Creation

advanced

AlphaShapeVertex_2 v;	default constructor.
AlphaShapeVertex_2 v(Point p);	constructor setting the point.
AlphaShapeVertex_2 v(Point p, Face_handle ff);	constructor setting the point associated to and an incident face.

advanced

Access Functions

std::pair< FT, FT >	*v.get_range()*	returns two alpha values $\alpha_1 \leq \alpha_2$, such as for α between α_1 and α_2, the vertex is attached but singular, and for α upper α_2, the vertex is regular.

Modifiers

advanced

void	*v.set_range(std::pair< FT, FT > I)*	sets the alpha values $\alpha_1 \leq \alpha_2$, such as for α between α_1 and α_2, the vertex is attached but singular, and for α upper α_2, the vertex is regular.

advanced

Class

CGAL::Alpha_shape_vertex_base_2< Traits, Vb, ExactAlphaComparisonTag>

Definition

The class *Alpha_shape_vertex_base_2< Traits, Vb, ExactAlphaComparisonTag>* is the default model for the concept *AlphaShapeVertex_2*.

#include <CGAL/Alpha_shape_vertex_base_2.h>

Parameters

The template parameter *Traits* has to be a model of *AlphaShapeTraits_2*.

The template parameter *Vb* has to be a model of *TriangulationVertexBase_2* (or *RegularTriangulationVertexBase_2*) if *Alpha_shape_vertex_base_2< Traits, Vb, ExactAlphaComparisonTag>* is intended to be used with an alpha-shape class based on a *Delaunay_triangulation_2* (or a *Regular_triangulation_2*).

The template parameter *ExactAlphaComparisonTag* is a tag that, when set to *CGAL::Tag_true*, triggers exact comparisons between alpha values. See the description provided in the documentation of *Alpha_shape_2* for more details. The default value is *CGAL::Tag_false*.

Is Model for the Concepts

AlphaShapeVertex_2

Inherits From

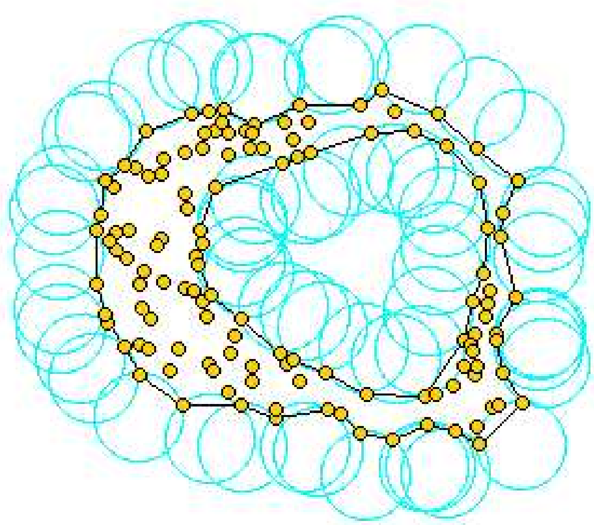
Vb

Chapter 43

3D Alpha Shapes

Tran Kai Frank Da, Sébastien Loriot, and Mariette Yvinec

Contents



Assume we are given a set S of points in 2D or 3D and we'd like to have something like "the shape formed by these points." This is quite a vague notion and there are probably many possible interpretations, the alpha shape being one of them. Alpha shapes can be used for shape reconstruction from a dense unorganized set of data points. Indeed, an alpha shape is demarcated by a frontier, which is a linear approximation of the original shape [BB97].

As mentioned in Edelsbrunner's and Mücke's paper [EM94], one can intuitively think of an alpha shape as the following. Imagine a huge mass of ice-cream making up the space $\mathbb{R}^3$ and containing the points as "hard" chocolate pieces. Using one of those sphere-formed ice-cream spoons we carve out all parts of the ice-cream block we can reach without bumping into chocolate pieces, thereby even carving out holes in the inside (e.g. parts not reachable by simply moving the spoon from the outside). We will eventually end up with a (not necessarily convex) object bounded by caps, arcs and points. If we now straighten all "round" faces to triangles and line segments, we have an intuitive description of what is called the alpha shape of S. Here's an example for this process in 2D (where our ice-cream spoon is simply a circle):

Alpha shapes depend on a parameter α from which they are named. What is α in the ice-cream game? α is the squared radius of the carving spoon. A very small value will allow us to eat up all of the ice-cream except the chocolate points themselves. Thus we already see that the alpha shape degenerates to the point-set S for $\alpha \to 0$. On the other hand, a huge value of α will prevent us even from moving the spoon between two points since it's way too large. So we will never spoon up ice-cream lying in the inside of the convex hull of S, and hence the alpha shape for $\alpha \to \infty$ is the convex hull of S.[1]

43.1 Definitions

More precisely, the definition of alpha shapes is based on an underlying triangulation that may be a Delaunay triangulation in case of basic alpha shapes or a regular triangulation (cf. 39.3) in case of weighted alpha shapes.

Let us consider the basic case with a Delaunay triangulation. We first define the alpha complex of the set of points S. The alpha complex is a subcomplex of the Delaunay triangulation. For a given value of α, the alpha complex includes all the simplices in the Delaunay triangulation which have an empty circumscribing sphere with squared radius equal or smaller than α. Here "empty" means that the open sphere do not include any points of S. The alpha shape is then simply the domain covered by the simplices of the alpha complex (see [EM94]).

In general, an alpha complex is a disconnected and non-pure complex: This means in particular that the alpha complex may have singular faces. For $0 \leq k \leq d-1$, a k-simplex of the alpha complex is said to be singular if it is not a facet of a $(k+1)$-simplex of the complex. CGAL provides two versions of alpha shapes. In the general mode, the alpha shapes correspond strictly to the above definition. The regularized mode provides a regularized version of the alpha shapes. It corresponds to the domain covered by a regularized version of the alpha complex where singular faces are removed (See Figure 43.1 for an example).

The alpha shapes of a set of points S form a discrete family, even though they are defined for all real numbers α. The entire family of alpha shapes can be represented through the underlying triangulation of S. In this representation each k-simplex of the underlying triangulation is associated with an interval that specifies for which values of α the k-simplex belongs to the alpha complex. Relying on this fact, the family of alpha shapes can be computed efficiently and relatively easily. Furthermore, we can select the optimal value of α to get an alpha shape including all data points and having less than a given number of connected components. Also, the alpha-values allows to define a filtration on the faces of the triangulation of a set of points. In this filtration, the faces of the triangulation are output in increasing order of the alpha value for which they appear in the alpha complex. In case of equal alpha value lower dimensional faces are output first.

[1] ice cream, ice cream!!! The wording of this introductory paragraphs is borrowed from Kaspar Fischer's " Introduction to Alpha Shapes" which can be found at http://people.inf.ethz.ch/fischerk/pubs/as.pdf. The picture has been taken from Walter Luh's homepage at http://www.stanford.edu/w̃luh/cs448b/alphashapes.html.

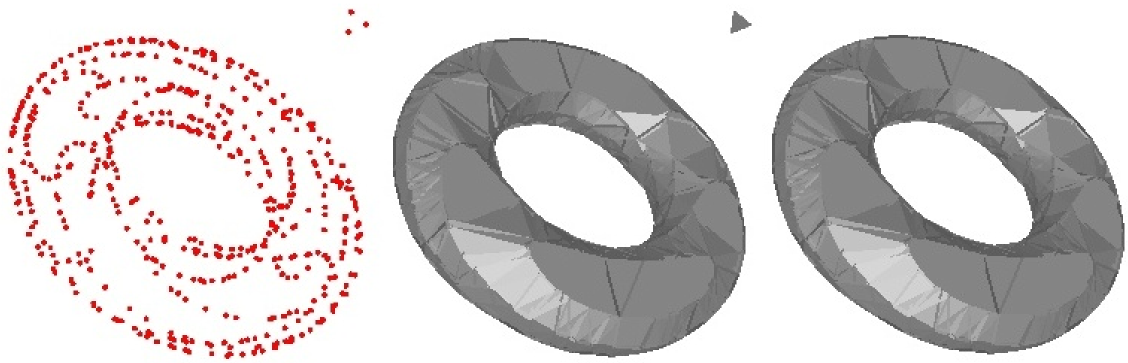


Figure 43.1: Comparison of general and regularized alpha-shape. **Left:** Some points are taken on the surface of a torus, three points being taken relatively far from the surface of the torus; **Middle:** The general alpha-shape (for a large enough alpha value) contains the singular triangle facet of the three isolated points; **Right:** The regularized version (for the same value of alpha) does not contains any singular facet.

The definition is analog in the case of weighted alpha shapes. The input set is now a set of weighted points (which can be regarded as spheres) and the underlying triangulation is the regular triangulation of this set. Two spheres, or two weighted points , with centers C_1, C_2 and radii r_1, r_2 are said to be orthogonal iff $C_1C_2^2 = r_1^2 + r_2^2$ and suborthogonal iff $C_1C_2^2 < r_1^2 + r_2^2$. For a given value of α the weighted alpha complex is formed with the simplices of the regular triangulation triangulation such that there is a sphere orthogonal to the weighted points associated with the vertices of the simplex and suborthogonal to all the other input weighted points. Once again the alpha shape is then defined as the domain covered by a the alpha complex and comes in general and regularized versions.

43.2 Functionality

43.2.1 Family of alpha shapes

The class *CGAL::Alpha_shape_3<Dt,ExactAlphaComparisonTag>* represents the whole family of alpha shapes for a given set of points. The class includes the underlying triangulation *Dt* of the set, and associates to each *k*-face of this triangulation an interval specifying for which values of α the face belongs to the alpha complex.

The class provides functions to set and get the current α-value, as well as an iterator that enumerates the α values where the alpha shape changes.

Also the class has a filtration member function that, given an output iterator with *CGAL::object* as value type, outputs the faces of the triangulation according to the order of apparition in the alpha complex when alpha increases.

Finally, it provides a function to determine the smallest value α such that the alpha shape satisfies the following two properties
(ii) all data points are either on the boundary or in the interior of the regularized version of the alpha shape (no singular faces).
(i) The number of components is equal or less than a given number .

The current implementation is static, that is after its construction points cannot be inserted or removed.

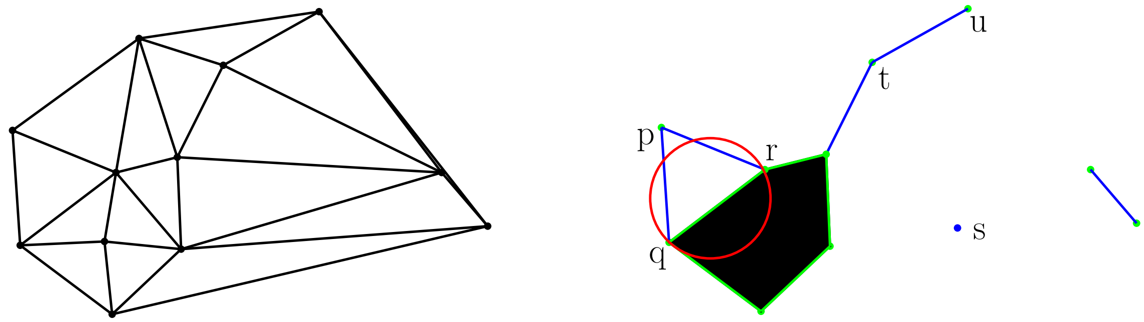


Figure 43.2: Classification of simplices, a 2D example. **Left:** The 2D Delaunay triangulation of a set of points; **Right:** Classification of simplices for a given alpha (the squared radius of the red circle). *INTERIOR*, *REGULAR* and *SINGULAR* simplices are depicted in black, green and blue respectively. *EXTERIOR* simplices are not depicted. The vertex s and the edge tu are *SINGULAR* since all higher dimension simplices they are incident to are *EXTERIOR*. The facet pqr is *EXTERIOR* because the squared radius of its circumscribed circle is larger than alpha.

43.2.2 Alpha shape for a fixed alpha

Given a value of alpha, the class *CGAL::Fixed_alpha_shape_3<Dt>* represents one alpha shape for a given set of points. The class includes the underlying triangulation *Dt* of the set, and associates to each *k*-face of this triangulation a classification type.

The current implementation of this class is dynamic, that is after its construction points can be inserted or removed.

43.2.3 Classification and iterators

Both classes provide member functions to classify for a (given) value of $alpha$ the different faces of the triangulation as *EXTERIOR*, *SINGULAR*, *REGULAR* or *INTERIOR* with respect to the alpha shape. A *k*-face on the boundary of the alpha complex is said to be: *REGULAR* if it is a subface of the alpha-complex which is a subface of a $(k+1)$-face of the alpha complex, and *SINGULAR* otherwise. A *k*-face of the alpha complex which is not on the boundary of the alpha complex is said to be *INTERIOR*. See Figure 43.2 for a 2D illustration.

The classes provide also output iterators to get for a given $alpha$ value the vertices, edges, facets and cells of the different types (*EXTERIOR*, *SINGULAR*, *REGULAR* or *INTERIOR*).

43.3 Concepts and Models

We currently do not specify concepts for the underlying triangulation type. Models that work for a familly alpha-shapes are the instantiations of the classes *CGAL::Delaunay_triangulation_3* and *CGAL::Periodic_3_Delaunay_triangulation_3* (see example 43.5.6). A model that works for a fixed alpha-shape are the instantiations of the class *CGAL::Delaunay_triangulation_3*. A model that works for a weighted alpha-shape is the class *CGAL::Regular_triangulation_3*. The triangulation needs a geometric traits class and a triangulation data structure as template parameters.

For the class *CGAL::Alpha_shape_3<Dt,ExactAlphaComparisonTag>*, the requirements of the traits class are described in the concepts *CGAL::AlphaShapeTraits_3* in the non-weighted case and

CGAL::WeightedAlphaShapeTraits_3 in the weighted case. The CGAL kernels are models in the non-weighted case, and the class *CGAL::Regular_triangulation_euclidean_traits_3* is a model in the weighted case. The triangulation data structure of the triangulation has to be a model of the concept *CGAL::TriangulationDataStructure_3*, and it must be parameterized with vertex and cell classes, which are model of the concepts *AlphaShapeVertex_3* and *AlphaShapeCell_3*. The package provides by default the classes *CGAL::Alpha_shape_vertex_base_3<Gt>* and *CGAL::Alpha_shape_cell_base_3<Gt>*. When using *CGAL::Periodic_3_Delaunay_triangulation_3* as underlying triangulation the vertex and cell classes need to be models to both *AlphaShapeVertex_3* and *Periodic_3TriangulationDSVertexBase_3* as well as *AlphaShapeCell_3* and *Periodic_3TriangulationDSCellBase_3* (see example 43.5.6).

For the class *CGAL::Fixed_alpha_shape_3<Dt>*, the requirements of the traits class are described in the concepts *CGAL::FixedAlphaShapeTraits_3* in the non-weighted case and *CGAL::FixedWeightedAlphaShapeTraits_3* in the weighted case. The CGAL kernels are models in the non-weighted case, and the class *CGAL::Regular_triangulation_euclidean_traits_3* is a model in the weighted case. The triangulation data structure of the triangulation has to be a model of the concept *CGAL::TriangulationDataStructure_3*, and it must be parameterized with vertex and cell classes, which are model of the concepts *FixedAlphaShapeVertex_3* and *FixedAlphaShapeCell_3*. The package provides models *CGAL::Fixed_alpha_shape_vertex_base_3<Gt>* and *CGAL::Fixed_alpha_shape_cell_base_3<Gt>*, respectively.

43.4 *Alpha_shape_3* or *Fixed_alpha_shape_3*

The class *CGAL::Alpha_shape_3<Dt,ExactAlphaComparisonTag>* represents the whole family of alpha shapes for a given set of points while the class *CGAL::Fixed_alpha_shape_3<Dt>* represents only one alpha shape (for a fixed alpha). When using the same kernel, *CGAL::Fixed_alpha_shape_3<Dt>* being a lighter version, it is naturally much more efficient when the alpha-shape is needed for a single given value of alpha. In addition note that the class *CGAL::Alpha_shape_3<Dt,ExactAlphaComparisonTag>* requires constructions (squared radius of simplices) while the class *CGAL::Fixed_alpha_shape_3<Dt>* uses only predicates. This implies that a certified construction of one (several) alpha-shape, using the *CGAL::Alpha_shape_3<Dt,ExactAlphaComparisonTag>* requires a kernel with exact predicates and exact constructions (or setting *ExactAlphaComparisonTag* to *CGAL::Tag_true*) while using a kernel with exact predicates is sufficient for the class *CGAL::Fixed_alpha_shape_3<Dt>*. This makes the class *CGAL::Fixed_alpha_shape_3<Dt>* even more efficient in this setting. In addition, note that the *Fixed* version is the only of the two that supports incremental insertion and removal of points.

We give the time spent while computing the alpha shape of a protein (considered as a set of weighted points) featuring 4251 atoms (using *gcc 4.3* under Linux with *-O3* and *-DNDEBUG* flags, on a 2.27GHz Intel(R) Xeon(R) E5520 CPU): Using *CGAL::Exact_predicates_inexact_constructions_kernel*, building the regular triangulation requires 0.09s, then the class *CGAL::Fixed_alpha_shape_3<Dt>* required 0.05s while the class *CGAL::Alpha_shape_3<Dt,ExactAlphaComparisonTag>* requires 0.35s if *ExactAlphaComparisonTag* is *CGAL::Tag_false* (and 0.70s with *CGAL::Tag_true*). Using *CGAL::Exact_predicates_exact_constructions_kernel*, building the regular triangulation requires 0.19s and then the class *CGAL::Alpha_shape_3<Dt,ExactAlphaComparisonTag>* requires 0.90s.

43.5 Examples

43.5.1 Example for Basic Alpha-Shapes

This example builds a basic alpha shape using a Delaunay triangulation as underlying triangulation.

```
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/Delaunay_triangulation_3.h>
#include <CGAL/Alpha_shape_3.h>

#include <fstream>
#include <list>
#include <cassert>

typedef CGAL::Exact_predicates_inexact_constructions_kernel Gt;

typedef CGAL::Alpha_shape_vertex_base_3<Gt>          Vb;
typedef CGAL::Alpha_shape_cell_base_3<Gt>            Fb;
typedef CGAL::Triangulation_data_structure_3<Vb,Fb>  Tds;
typedef CGAL::Delaunay_triangulation_3<Gt,Tds>       Triangulation_3;
typedef CGAL::Alpha_shape_3<Triangulation_3>         Alpha_shape_3;

typedef Gt::Point_3                                  Point;
typedef Alpha_shape_3::Alpha_iterator               Alpha_iterator;

int main()
{
  std::list<Point> lp;

  //read input
  std::ifstream is("./data/bunny_1000");
  int n;
  is >> n;
  std::cout << "Reading " << n << " points " << std::endl;
  Point p;
  for( ; n>0 ; n--)    {
    is >> p;
    lp.push_back(p);
  }

  // compute alpha shape
  Alpha_shape_3 as(lp.begin(),lp.end());
  std::cout << "Alpha shape computed in REGULARIZED mode by default"
            << std::endl;

  // find optimal alpha value
  Alpha_iterator opt = as.find_optimal_alpha(1);
  std::cout << "Optimal alpha value to get one connected component is "
            <<  *opt    << std::endl;
  as.set_alpha(*opt);
  assert(as.number_of_solid_components() == 1);
  return 0;
}
```

File: examples/Alpha_shapes_3/ex_alpha_shapes_3.cpp

43.5.2 Building Basic Alpha Shapes for Many Points

When many points are input in the alpha shape, say more than 10 000, it may pay off to use a Delaunay triangulation with *Fast_location* policy as underlying triangulation in order to speed up point location queries (cf. 39.4.3).

```
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/Delaunay_triangulation_3.h>
#include <CGAL/Alpha_shape_3.h>

#include <fstream>
#include <list>
#include <cassert>

typedef CGAL::Exact_predicates_inexact_constructions_kernel K;

typedef CGAL::Alpha_shape_vertex_base_3<K>               Vb;
typedef CGAL::Alpha_shape_cell_base_3<K>                 Fb;
typedef CGAL::Triangulation_data_structure_3<Vb,Fb>      Tds;
typedef CGAL::Delaunay_triangulation_3<K,Tds,CGAL::Fast_location>
   Delaunay;
typedef CGAL::Alpha_shape_3<Delaunay>                    Alpha_shape_3;

typedef K::Point_3                                  Point;
typedef Alpha_shape_3::Alpha_iterator               Alpha_iterator;
typedef Alpha_shape_3::NT                           NT;

int main()
{
  Delaunay dt;
  std::ifstream is("./data/bunny_1000");
  int n;
  is >> n;
  Point p;
  std::cout << n << " points read" << std::endl;
  for( ; n>0 ; n--) {
    is >> p;
    dt.insert(p);
  }
  std::cout << "Delaunay computed." << std::endl;

  // compute alpha shape
  Alpha_shape_3 as(dt);
  std::cout << "Alpha shape computed in REGULARIZED mode by defaut."
            << std::endl;

   // find optimal alpha values
  Alpha_shape_3::NT alpha_solid = as.find_alpha_solid();
  Alpha_iterator opt = as.find_optimal_alpha(1);
  std::cout << "Smallest alpha value to get a solid through data points
   is "
            << alpha_solid << std::endl;
  std::cout << "Optimal alpha value to get one connected component is "
            <<  *opt    << std::endl;
```

```
  as.set_alpha(*opt);
  assert(as.number_of_solid_components() == 1);
  return 0;
}
```

File: examples/Alpha_shapes_3/ex_alpha_shapes_with_fast_location_3.cpp

43.5.3 Example for Weighted Alpha-Shapes

The following examples build a weighted alpha shape requiring a regular triangulation as underlying triangulation. The alpha shape is built in *GENERAL* mode.

```
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/Regular_triangulation_euclidean_traits_3.h>
#include <CGAL/Regular_triangulation_3.h>
#include <CGAL/Alpha_shape_3.h>
#include <list>

typedef CGAL::Exact_predicates_inexact_constructions_kernel K;

typedef CGAL::Regular_triangulation_euclidean_traits_3<K> Gt;

typedef CGAL::Alpha_shape_vertex_base_3<Gt>         Vb;
typedef CGAL::Alpha_shape_cell_base_3<Gt>           Fb;
typedef CGAL::Triangulation_data_structure_3<Vb,Fb> Tds;
typedef CGAL::Regular_triangulation_3<Gt,Tds>       Triangulation_3;
typedef CGAL::Alpha_shape_3<Triangulation_3>        Alpha_shape_3;

typedef Alpha_shape_3::Cell_handle          Cell_handle;
typedef Alpha_shape_3::Vertex_handle        Vertex_handle;
typedef Alpha_shape_3::Facet                Facet;
typedef Alpha_shape_3::Edge                 Edge;
typedef Gt::Weighted_point                  Weighted_point;
typedef Gt::Bare_point                      Bare_point;

int main()
{
  std::list<Weighted_point> lwp;

  //input : a small molecule
  lwp.push_back(Weighted_point(Bare_point( 1, -1, -1), 4));
  lwp.push_back(Weighted_point(Bare_point(-1,  1, -1), 4));
  lwp.push_back(Weighted_point(Bare_point(-1, -1,  1), 4));
  lwp.push_back(Weighted_point(Bare_point( 1,  1,  1), 4));
  lwp.push_back(Weighted_point(Bare_point( 2,  2,  2), 1));

  //build alpha_shape  in GENERAL mode and set alpha=0
  Alpha_shape_3  as(lwp.begin(), lwp.end(), 0, Alpha_shape_3::GENERAL);

  //explore the 0-shape - It is dual to the boundary of the union.
  std::list<Cell_handle> cells;
  std::list<Facet>       facets;
```

```
  std::list<Edge>        edges;
  as.get_alpha_shape_cells(std::back_inserter(cells),
                           Alpha_shape_3::INTERIOR);
  as.get_alpha_shape_facets(std::back_inserter(facets),
                            Alpha_shape_3::REGULAR);
  as.get_alpha_shape_facets(std::back_inserter(facets),
                            Alpha_shape_3::SINGULAR);
  as.get_alpha_shape_edges(std::back_inserter(edges),
                           Alpha_shape_3::SINGULAR);
  std::cout << " The 0-shape has : " << std::endl;
  std::cout << cells.size() << " interior tetrahedra" << std::endl;
  std::cout << facets.size() << " boundary facets" << std::endl;
  std::cout << edges.size()  << " singular edges" << std::endl;
  return 0;
}
```

File: examples/Alpha_shapes_3/ex_weighted_alpha_shapes_3.cpp

43.5.4 Example for Fixed Weighted Alpha-Shapes

Same example as previous but using a fixed value of alpha.

```
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/Regular_triangulation_3.h>
#include <CGAL/Regular_triangulation_euclidean_traits_3.h>
#include <CGAL/Fixed_alpha_shape_3.h>
#include <CGAL/Fixed_alpha_shape_vertex_base_3.h>
#include <CGAL/Fixed_alpha_shape_cell_base_3.h>
#include <list>

typedef CGAL::Exact_predicates_inexact_constructions_kernel K;

typedef CGAL::Regular_triangulation_euclidean_traits_3<K> Gt;

typedef CGAL::Fixed_alpha_shape_vertex_base_3<Gt>         Vb;
typedef CGAL::Fixed_alpha_shape_cell_base_3<Gt>           Fb;
typedef CGAL::Triangulation_data_structure_3<Vb,Fb>       Tds;
typedef CGAL::Regular_triangulation_3<Gt,Tds>
   Triangulation_3;
typedef CGAL::Fixed_alpha_shape_3<Triangulation_3>
   Fixed_alpha_shape_3;

typedef Fixed_alpha_shape_3::Cell_handle                  Cell_handle;
typedef Fixed_alpha_shape_3::Vertex_handle                Vertex_handle;
typedef Fixed_alpha_shape_3::Facet                        Facet;
typedef Fixed_alpha_shape_3::Edge                         Edge;
typedef Gt::Weighted_point                                Weighted_point;
typedef Gt::Bare_point                                    Bare_point;

int main()
{
  std::list<Weighted_point> lwp;
```

```
  //input : a small molecule
  lwp.push_back(Weighted_point(Bare_point( 1, -1, -1), 4));
  lwp.push_back(Weighted_point(Bare_point(-1,  1, -1), 4));
  lwp.push_back(Weighted_point(Bare_point(-1, -1,  1), 4));
  lwp.push_back(Weighted_point(Bare_point( 1,  1,  1), 4));
  lwp.push_back(Weighted_point(Bare_point( 2,  2,  2), 1));

  //build one alpha_shape  with alpha=0
  Fixed_alpha_shape_3  as(lwp.begin(), lwp.end(), 0);

  //explore the 0-shape - It is dual to the boundary of the union.
  std::list<Cell_handle> cells;
  std::list<Facet>       facets;
  std::list<Edge>        edges;
  as.get_alpha_shape_cells(std::back_inserter(cells),
                           Fixed_alpha_shape_3::INTERIOR);
  as.get_alpha_shape_facets(std::back_inserter(facets),
                            Fixed_alpha_shape_3::REGULAR);
  as.get_alpha_shape_facets(std::back_inserter(facets),
                            Fixed_alpha_shape_3::SINGULAR);
  as.get_alpha_shape_edges(std::back_inserter(edges),
                           Fixed_alpha_shape_3::SINGULAR);
  std::cout << " The 0-shape has : " << std::endl;
  std::cout << cells.size() << " interior tetrahedra" << std::endl;
  std::cout << facets.size() << " boundary facets" << std::endl;
  std::cout << edges.size()  << " singular edges" << std::endl;
  return 0;
}
```

File: examples/Alpha_shapes_3/ex_fixed_weighted_alpha_shapes_3.cpp

43.5.5 Building an Alpha Shapes with Exact Comparisons of Alpha Values

On some platforms, the alpha shapes of the set of points of this example cannot be computed when using a traits with inexact constructions. To be able to compute them with a traits with inexact constructions, the tag *ExactAlphaComparisonTag* is set to *CGAL::Tag_true*.

```
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/Delaunay_triangulation_3.h>
#include <CGAL/Alpha_shape_3.h>

#include <fstream>
#include <list>
#include <cassert>

typedef CGAL::Exact_predicates_inexact_constructions_kernel
        Gt;
typedef CGAL::Tag_true
        Alpha_cmp_tag;
//We use CGAL::Default to skip the complete declaration of base classes
```

```
typedef CGAL::Alpha_shape_vertex_base_3<Gt,CGAL::Default,Alpha_cmp_tag>
        Vb;
typedef CGAL::Alpha_shape_cell_base_3<Gt,CGAL::Default,Alpha_cmp_tag>
        Fb;
typedef CGAL::Triangulation_data_structure_3<Vb,Fb>
        Tds;
typedef CGAL::Delaunay_triangulation_3<Gt,Tds>
        Triangulation_3;
//Alpha shape with ExactAlphaComparisonTag set to true (note that the tag
    is also
//set to true for Vb and Fb)
typedef CGAL::Alpha_shape_3<Triangulation_3,Alpha_cmp_tag>
        Alpha_shape_3;
typedef Gt::Point_3
        Point;

int main()
{
  //Set of points for which the alpha shapes cannot be computed with
  //a floating point alpha value (on certain platforms)
  std::list<Point> lp;
  lp.push_back(Point(49.2559,29.1767,6.7723));
  lp.push_back(Point(49.3696,31.4775,5.33777));
  lp.push_back(Point(49.4264,32.6279,4.6205));
  lp.push_back(Point(49.3127,30.3271,6.05503));


   // compute alpha shape
  Alpha_shape_3 as(lp.begin(),lp.end(),0,Alpha_shape_3::GENERAL);

  return 0;
}
```

File: examples/Alpha_shapes_3/ex_alpha_shapes_exact_alpha.cpp

43.5.6 Example for periodic Alpha Shapes

The following example shows how to use the periodic Delaunay triangulation (Chapter 41) as underlying triangulation for the alpha shape computation.

In order to define the original domain and to benefit from the built-in heuristic optimizations of the periodic Delaunay triangulation computation it is recommended to first construct the triangulation and then construct the alpha shape from it. The alpha shape constructor that takes a point range can be used as well but in this case the original domain cannot be specified and the default unit cube will be chosen and no optimizations will be used.

It is also recommended to switch the triangulation to 1-sheeted covering if possible. Note that a periodic triangulation in 27-sheeted covering space is degenerate. In this case an exact constructions kernel needs to be used to compute the alpha shapes. Otherwise the results will suffer from round-off problems.

```
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/Periodic_3_triangulation_traits_3.h>
#include <CGAL/Periodic_3_Delaunay_triangulation_3.h>
```

```
#include <CGAL/Alpha_shape_3.h>

#include <CGAL/Random.h>
#include <CGAL/point_generators_3.h>

// Traits
typedef CGAL::Exact_predicates_inexact_constructions_kernel K;
typedef CGAL::Periodic_3_triangulation_traits_3<K> PK;

// Vertex type
typedef CGAL::Periodic_3_triangulation_ds_vertex_base_3<> DsVb;
typedef CGAL::Triangulation_vertex_base_3<PK,DsVb> Vb;
typedef CGAL::Alpha_shape_vertex_base_3<PK,Vb> AsVb;
// Cell type
typedef CGAL::Periodic_3_triangulation_ds_cell_base_3<> DsCb;
typedef CGAL::Triangulation_cell_base_3<PK,DsCb> Cb;
typedef CGAL::Alpha_shape_cell_base_3<PK,Cb> AsCb;

typedef CGAL::Triangulation_data_structure_3<AsVb,AsCb> Tds;
typedef CGAL::Periodic_3_Delaunay_triangulation_3<PK,Tds> P3DT3;
typedef CGAL::Alpha_shape_3<P3DT3>  Alpha_shape_3;

typedef PK::Point_3                                 Point;

int main()
{
  typedef CGAL::Creator_uniform_3<double, Point> Creator;
  CGAL::Random random(7);
  CGAL::Random_points_in_cube_3<Point, Creator> in_cube(1, random);
  std::vector<Point> pts;

  // Generating 1000 random points
  for (int i=0 ; i < 1000 ; i++) {
    Point p = *in_cube++;
    pts.push_back(p);
  }

  // Define the periodic cube
  P3DT3 pdt(PK::Iso_cuboid_3(-1,-1,-1,1,1,1));
  // Heuristic for inserting large point sets (if pts is reasonably
   large)
  pdt.insert(pts.begin(), pts.end(), true);
  // As pdt won't be modified anymore switch to 1-sheeted cover if
   possible
  if (pdt.is_triangulation_in_1_sheet())
   pdt.convert_to_1_sheeted_covering();
  std::cout << "Periodic Delaunay computed." << std::endl;

  // compute alpha shape
  Alpha_shape_3 as(pdt);
  std::cout << "Alpha shape computed in REGULARIZED mode by default."
            << std::endl;

   // find optimal alpha values
```

```
  Alpha_shape_3::NT alpha_solid = as.find_alpha_solid();
  Alpha_shape_3::Alpha_iterator opt = as.find_optimal_alpha(1);
  std::cout << "Smallest alpha value to get a solid through data points
     is "
            << alpha_solid << std::endl;
  std::cout << "Optimal alpha value to get one connected component is "
            <<  *opt    << std::endl;
  as.set_alpha(*opt);
  assert(as.number_of_solid_components() == 1);
  return 0;
}
```

File: examples/Alpha_shapes_3/ex_periodic_alpha_shapes_3.cpp

3D Alpha Shapes Reference Manual

Tran Kai Frank Da, Sébastien Loriot, and Mariette Yvinec

Alpha shapes definition is based on an underlying triangulation that may be a Delaunay triangulation in case of basic alpha shapes or a regular triangulation (cf. 39.3) in case of weighted alpha shapes.

Let us consider the basic case with a Delaunay triangulation. We first define the alpha complex of the set of points S. The alpha complex is a subcomplex of the Delaunay triangulation. For a given value of α, the alpha complex includes all the simplices in the Delaunay triangulation which have an empty circumscribing sphere with squared radius equal or smaller than α. Here "empty" means that the open sphere do not include any points of S. The alpha shape is then simply the domain covered by the simplices of the alpha complex (see [EM94]).

In general, an alpha complex is a non-connected and non-pure complex. This means in particular that the alpha complex may have singular faces. For $0 \leq k \leq d-1$, a k-simplex of the alpha complex is said to be singular if it is not a facet of a $(k+1)$-simplex of the complex

The alpha shapes of a set of points S form a discrete family, even though they are defined for all real numbers α. The entire family of alpha shapes can be represented through the underlying triangulation of S. In this representation each k-simplex of the underlying triangulation is associated with an interval that specifies for which values of α the k-simplex belongs to the alpha complex. Relying on this fact, the family of alpha shapes can be computed efficiently and relatively easily. Furthermore, we can select the optimal value of α to get an alpha shape including all data points and having less than a given number of connected components.

The definition is analog in the case of weighted alpha shapes. The input set is now a set of weighted points (which can be regarded as spheres) and the underlying triangulation is the regular triangulation of this set. Two spheres, or two weighted points , with centers C_1, C_2 and radii r_1, r_2 are said to be orthogonal iff $C_1C_2^2 = r_1^2 + r_2^2$ and suborthogonal iff $C_1C_2^2 < r_1^2 + r_2^2$. For a given value of α the weighted alpha complex is formed with the simplices of the regular triangulation triangulation such that there is a sphere orthogonal to the weighted points associated with the vertices of the simplex and suborthogonal to all the other input weighted points. Once again the alpha shape is then defined as the domain covered by a the alpha complex.

CGAL provides two versions to compute alpha shapes. The first one gives access to an explicit classification of all the simplices for a fixed alpha value. The second one gives access to the entire family of alpha shapes of a set of points. This latter version comes with two modes. In the general mode, the alpha shapes correspond strictly to the definition previously made. The regularized mode provides a regularized version of the alpha shapes corresponding to the domain covered by a regularized version of the alpha complex where singular faces are removed.

43.6 Classified Reference Pages

Concepts

Classes

43.7 Alphabetical List of Reference Pages

AlphaShapeCell_3

Concept

Definition

This concept describes the requirements for the base cell of an alpha shape.

Refines

TriangulationCellBase_3.

AlphaShapeCell_3:: NT	A number type. Must be the same as the number type used in the traits class of the triangulation underlying the alpha shape.
AlphaShapeCell_3:: Alpha_status_iterator	An iterator with value type *CGAL::Alpha_status<NT>*.

Creation

AlphaShapeCell_3 f; — default constructor.

AlphaShapeCell_3 f(Vertex_handle v0, Vertex_handle v1, Vertex_handle v2, Vertex_handle v3);

constructor setting the incident vertices.

AlphaShapeCell_3 f(Vertex_handle v0, Vertex_handle v1, Vertex_handle v2, Vertex_handle v3, Cell_handle n0, Cell_handle n1, Cell_handle n2, Cell_handle n3)

constructor setting the incident vertices and the neighboring cells.

Access Functions

NT *f.get_alpha()* — Returns the alpha value of the cell.

Alpha_status_iterator

f.get_facet_status(int i)

Returns an iterator on the *CGAL::Alpha_status<NT>* of the facet i of the cell.

Modifiers

void	*f.set_alpha(NT alpha)*	
		Sets the critical value of the cell.
void	*f.set_facet_status(int i, Alpha_status_iterator as)*	
		Sets the iterator pointing to the *CGAL::Alpha_status<NT>* of the facet *i* of the cell.

See Also

CGAL::Alpha_status

AlphaShapeTraits_3

Concept

Definition

The concept AlphaShapeTraits_3 describes the requirements for the geometric traits class of the underlying Delaunay triangulation of a basic alpha shape.

Refines

DelaunayTriangulationTraits_3

In addition to the requirements described in the concept *DelaunayTriangulationTraits_3*, the geometric traits class of a Delaunay triangulation plugged in a basic alpha shapes provides the following.

Types

AlphaShapeTraits_3:: FT	A number type compatible with the type used for the points coordinate.
AlphaShapeTraits_3:: Compute_squared_radius_3	An object constructor able to compute: the squared radius of the smallest circumscribing sphere of 4 points *p0, p1, p2, p3*, the squared radius of the smallest circumscribing sphere of 3 points *p0, p1, p2*, the squared radius of the smallest circumscribing sphere of 2 points *p0, p1*, and also the squared radius of the smallest circumscribing sphere to a single point *p0* (which is *FT(0)*).

Creation

AlphaShapeTraits_3 traits;	Default constructor.

Access Functions

Compute_squared_radius_3 *traits.compute_squared_radius_3_object()*

Has Models

All *CGAL* kernels.
CGAL::Exact_predicates_inexact_constructions_kernel (recommended)
CGAL::Exact_predicates_exact_constructions_kernel
CGAL::Filtered_kernel
CGAL::Cartesian
CGAL::Simple_cartesian
CGAL::Homogeneous
CGAL::Simple_homogeneous

Concept

AlphaShapeVertex_3

Definition

This concept describe the requirements for the base vertex of an alpha shape.

Refines

TriangulationVertexBase_3.

Types

AlphaShapeVertex_3:: Point	Must be the same as the point type provided by the geometric traits class of the triangulation.
AlphaShapeVertex_3:: Alpha_status	Must be *CGAL::Alpha_status<NT>* where *NT* is the number type used in the geometric traits class of the triangulation.

Creation

AlphaShapeVertex_3 v;	default constructor.
AlphaShapeVertex_3 v(Point p);	constructor setting the point associated to.
AlphaShapeVertex_3 v(Point p, Cell_handle c);	constructor setting the point associated to and an incident cell.

Modifiers

*Alpha_status**	*v.get_alpha_status()*	Returns a pointer the alpha status of the vertex.

See Also

CGAL::Alpha_status

CGAL::Alpha_shape_3<Dt,ExactAlphaComparisonTag>

Class

Definition

The class *Alpha_shape_3<Dt,ExactAlphaComparisonTag>* represents the family of alpha shapes of points in the 3D space for *all* real α. It maintains an underlying triangulation of the class *Dt*. Each k-dimensional face of Dt is associated with an interval that specifies for which values of alpha the face belongs to the alpha shape. The second template parameter *ExactAlphaComparisonTag* is a tag that, when set to *CGAL::Tag_true*, triggers exact comparisons between alpha values. This is useful when the Delaunay triangulation is instantiated with an exact predicates inexact constructions kernel. By default the *ExactAlphaComparisonTag* is set to *CGAL::Tag_false* as it induces a small overhead. Note that since such a strategy does not make sense if used together with a traits class with exact constructions, the tag *ExactAlphaComparisonTag* is not taken into account if *Dt::Geom_traits::FT* is not a floating point number type.

Note that this class is at the same time used for *basic* and for *weighted* Alpha Shapes.

#include <CGAL/Alpha_shape_3.h>

Inherits From

Dt

This class is the underlying triangulation class.

The modifying functions *insert* and *remove* will overwrite the inherited functions. At the moment, only the static version is implemented.

Types

Alpha_shape_3<Dt,ExactAlphaComparisonTag>:: Gt

the alpha shape traits type.

it has to derive from a triangulation traits class. For example *Dt::Point* is a Point class.

Alpha_shape_3<Dt,ExactAlphaComparisonTag>:: FT

the number type of alpha values.
In case *ExactAlphaComparisonTag* is *CGAL::Tag_false*, it is Gt::FT.
In case *ExactAlphaComparisonTag* is *CGAL::Tag_true*, it is a number type allowing filtered exact comparisons (that is, interval arithmetic is first used before resorting to exact arithmetic).
Access to the interval containing the exact value is provided through the function *FT::Approximate_nt approx() const* where *FT::Approximate_nt* is *Interval_nt<Protected>* with *Protected=true*.
Access to the exact value is provided through the function *FT::Exact_nt exact() const* where *FT::Exact_nt* depends on the configuration of CGAL (it is *CGAL::Gmpq* if *gmp* is available and *CGAL::Quotient< CGAL::MP_Float>* otherwise).
It must be noted that an object of type *FT* is valid as long as the alpha shapes class that creates it is valid and has not been modified. For convenience, classical comparison operators are provided for the type *FT*.

Alpha_shape_3<Dt,ExactAlphaComparisonTag>:: size_type

The size type.

Alpha_shape_3<Dt,ExactAlphaComparisonTag>:: Alpha_iterator

A bidirectional and non-mutable iterator that allow to traverse the increasing sequence of different alpha values.
Precondition: Its *value_type* is *FT*

enum Mode { *GENERAL, REGULARIZED*}*;*

In GENERAL mode, the alpha complex can have singular faces, i. e. faces of dimension k, for $k = (0,1,2)$ that are not subfaces of a $k+1$ face of the complex. In REGULARIZED mode, the complex is regularized, that is singular faces are dropped and the alpha complex includes only a subset of the tetrahedral cells of the triangulation and the subfaces of those cells.

enum Classification_type { *EXTERIOR, SINGULAR, REGULAR, INTERIOR*}*;*

Enum to classify the faces of the underlying triangulation with respect to the alpha shape.
In GENERAL mode, for $k = (0,1,2)$, each k-dimensional simplex of the triangulation can be classified as EXTERIOR, SINGULAR, REGULAR or INTERIOR. In GENERAL mode a k simplex is REGULAR if it is on the boundary f the alpha complex and belongs to a $k+1$ simplex in this complex and it is SINGULAR if it is a boundary simplex that is not included in a $k+1$ simplex of the complex.
In REGULARIZED mode, for $k = (0,1,2)$ each k-dimensional simplex of the triangulation can be classified as EXTERIOR, REGULAR or INTERIOR, i.e. there is no singular faces. A k simplex is REGULAR if it is on the boundary of alpha complex and belongs to a tetrahedral cell of the complex.

Creation

Alpha_shape_3<Dt,ExactAlphaComparisonTag> A(FT alpha = 0, Mode m = REGULARIZED);

Introduces an empty alpha shape data structure *A* for and set the current alpha value to *alpha* and the mode to *m*.

Alpha_shape_3<Dt,ExactAlphaComparisonTag> A(Dt& dt, FT alpha = 0, Mode m = REGULARIZED);

Build an alpha shape of mode *m* from the triangulation *dt*. Be careful that this operation destroys the triangulation.

template < class InputIterator >
Alpha_shape_3<Dt,ExactAlphaComparisonTag> A(InputIterator first, InputIterator last, FT alpha = 0, Mode m = REGULARIZED)

Build an alpha shape data structure in mode *m* for the points in the range [*first*, *last*) and set the current alpha value to *alpha*.
Precondition: The *value_type* of *first* and *last* is *Point* (the type point of the underlying triangulation.)

Modifiers

template < class InputIterator >
std::ptrdiff_t A.make_alpha_shape(InputIterator first, InputIterator last)

Initialize the alpha shape data structure for points in the range [*first*, *last*). Returns the number of data points inserted in the underlying triangulation.
If the function is applied to an non-empty alpha shape data structure, it is cleared before initialization.
Precondition: The *value_type* of *first* and *last* is *Point*.

void *A.clear()* Clears the structure.

FT *A.set_alpha(FT alpha)*

Sets the α-value to *alpha*. Returns the previous α-value.
Precondition: $alpha \geq 0$.

Mode *A.set_mode(Mode m = REGULARIZED)*

Sets *A* in GENERAL or REGULARIZED. Returns the previous mode. Changing the mode of an alpha shape data structure entails a partial re-computation of the data structure.

Query Functions

Mode *A.get_mode(void) const*

Returns whether *A* is general or regularized.

FT *A.get_alpha(void) const*

Returns the current α-value.

FT *A.get_nth_alpha(int n) const*

Returns the *n*-th alpha-value, sorted in an increasing order.
Precondition: *n* ¡ number of alphas.

size_type *A.number_of_alphas() const*

Returns the number of different alpha-values.

Classification_type

A.classify(Point p, FT alpha = get_alpha())

Locates a point *p* in the underlying triangulation and Classifies the associated k-face with respect to *alpha*.

Classification_type

A.classify(Cell_handle f, FT alpha = get_alpha())

Classifies the cell *f* of the underlying triangulation with respect to *alpha*.

Classification_type

A.classify(Facet f, FT alpha = get_alpha())

Classifies the facet *f* of the underlying triangulation with respect to *alpha*.

Classification_type

A.classify(Cell_handle f, int i, FT alpha = get_alpha())

Classifies the facet of the cell *f* opposite to the vertex with index *i* of the underlying triangulation with respect to *alpha*.

Classification_type

A.classify(Edge e, FT alpha = get_alpha())

Classifies the edge *e* with respect to *alpha* .

Classification_type

A.classify(Vertex_handle v, FT alpha = get_alpha())

Classifies the vertex *v* of the underlying triangulation with respect to *alpha*.

template<class OutputIterator>
OutputIterator

A.get_alpha_shape_cells(OutputIterator it, Classification_type type, FT alpha = get_alpha())

Write the cells which are of type *type* for the alpha value *alpha* to the sequence pointed to by the output iterator *it*. Returns past the end of the output sequence.

template<class OutputIterator>
OutputIterator

A.get_alpha_shape_facets(OutputIterator it,
Classification_type type,
FT alpha= get_alpha())

Write the facets which are of type *type* for the alpha value *alpha* to the sequence pointed to by the output iterator *it*. Returns past the end of the output sequence.

template<class OutputIterator>
OutputIterator

A.get_alpha_shape_edges(OutputIterator it,
Classification_type type,
FT alpha = get_alpha())

Write the edges which are of type *type* for the alpha value *alpha* to the sequence pointed to by the output iterator *it*. Returns past the end of the output sequence.

template<class OutputIterator>
OutputIterator

A.get_alpha_shape_vertices(OutputIterator it, Classification_type type, FT alpha)

Write the vertices which are of type *type* for the alpha value *alpha* to the sequence pointed to by the output iterator *it*. Returns past the end of the output sequence.

template<class OutputIterator>
OutputIterator

A.filtration(OutputIterator it)

Output all the faces of the triangulation in increasing order of the alpha value for which they appear in the alpha complex. In case of equal alpha value lower dimensional faces are output first. The value type of the OutputIterator has to be a CGAL::Object

Traversal of the α-Values

Alpha_iterator

A.alpha_begin() const

Returns an iterator that allows to traverse the sorted sequence of α-values of the family of alpha shapes.

Alpha_iterator

A.alpha_end() const

Returns the corresponding past-the-end iterator.

Alpha_iterator

A.alpha_find(FT alpha) const

Returns an iterator pointing to an element with α-value *alpha*, or the corresponding past-the-end iterator if such an element is not found.

Alpha_iterator

A.alpha_lower_bound(FT alpha) const

Returns an iterator pointing to the first element with α-value not less than *alpha*.

Alpha_iterator

A.alpha_upper_bound(FT alpha) const

Returns an iterator pointing to the first element with α-value greater than *alpha*.

Operations

size_type *A.number_of_solid_components(FT alpha = get_alpha())*

Returns the number of solid components of *A*, that is, the number of components of its regularized version.

Alpha_iterator

A.find_optimal_alpha(size_type nb_components) const

Returns an iterator pointing to smallest α value such that *A* satisfies the following two properties:
all data points are either on the boundary or in the interior of the regularized version of *A*.
The number of solid component of *A* is equal to or smaller than *nb_components*.

I/O

The I/O operators are defined for *iostream*, and for the window stream provided by CGAL. The format for the iostream is an internal format.

#include <CGAL/IO/io.h>

ostream& *ostream& os << A* Inserts the alpha shape *A* for the current alpha value into the stream *os*.
Precondition: The insert operator must be defined for *Point*.

#include <CGAL/IO/Geomview_stream.h>

#include <CGAL/IO/alpha_shape_geomview_ostream_3.h>

Geomview_stream&

Geomview_stream& W << A

Inserts the alpha shape *A* for the current alpha value into the Geomview stream *W*.
Precondition: The insert operator must be defined for *GT::Point* and *GT::Triangle*.

Implementation

In GENERAL mode, the alpha intervals of each triangulation face is computed and stored at initialization time. In REGULARIZED mode, the alpha shape intervals of edges are not stored nor computed at initialization. Edges are simply classified on the fly upon request. This allows to have much faster building of alpha shapes in REGULARIZED mode.

A.alpha find uses linear search, while *A.alpha lower bound* and *A.alpha upper bound* use binary search. *A.number of solid components* performs a graph traversal and takes time linear in the number of cells of the underlying triangulation. *A.find of optimal alpha* uses binary search and takes time $O(n \log n)$, where n is the number of points.

Class

CGAL::Alpha_shape_cell_base_3<Traits,Fb,ExactAlphaComparisonTag,WeightedTag>

Definition

The class *Alpha_shape_cell_base_3<Traits,Fb,ExactAlphaComparisonTag,WeightedTag>* is the default model for the concept *AlphaShapeCell_3*.

The class has four parameters. The traits class *Traits* is the one that is provided to the *Alpha_shape_3* class. The second parameter *Fb* is a base class instantiated by default with *CGAL::Triangulation_cell_base_3<Traits>* . The third parameter *ExactAlphaComparisonTag* is a tag that, when set to *CGAL::Tag_true*, triggers exact comparisons between alpha values. See the description provided in the documentation of *Alpha_shape_3* for more details. The default value is *CGAL::Tag_false*. The fourth parameter *WeightedTag* is used only if *ExactAlphaComparisonTag* is *CGAL::Tag_true*. It must be *CGAL::Tag_true* if the underlying triangulation of the alpha shape to be used is a regular triangulation and *CGAL::Tag_false* otherwise. The default is *CGAL::Tag_false*.

#include <CGAL/Alpha_shape_cell_base_3.h>

Is Model for the Concepts

AlphaShapeCell_3

Inherits From

Fb

CGAL::Alpha_shape_vertex_base_3<Traits,Vb,ExactAlphaComparisonTag,WeightedT

Definition

The class *Alpha_shape_vertex_base_3<Traits,Vb,ExactAlphaComparisonTag,WeightedTag>* is the default model for the concept *AlphaShapeVertex_3*.

The class has four parameters : the traits class *Traits* which provides the type for the points or the weighted points. The second parameter *Vb* is a base class instantiated by default with *CGAL::Triangulation_vertex_base_3<Traits>*. The third parameter *ExactAlphaComparisonTag* is a tag that, when set to *CGAL::Tag_true*, triggers exact comparisons between alpha values. See the description provided in the documentation of *Alpha_shape_3* for more details. The default value is *CGAL::Tag_false*. The fourth parameter *WeightedTag* is used only if *ExactAlphaComparisonTag* is *CGAL::Tag_true*. It must be *CGAL::Tag_true* if the underlying triangulation of the alpha shape to be used is a regular triangulation and *CGAL::Tag_false* otherwise. The default is *CGAL::Tag_false*.

#include <CGAL/Alpha_shape_vertex_base_3.h>

Is Model for the Concepts

AlphaShapeVertex_3

Inherits From

Vb

CGAL::Alpha_status<NT>

Definition

The class *Alpha_status<NT>* is a small data structure to store the critical alpha values of faces of an alpha shape. Each face has three critical alpha values, called *alpha_min*, *alpha_mid* and *alpha_max* in increasing order. The face will be exterior for any $\alpha < alpha\_min$, singular for $alpha\_min \leq \alpha < alpha\_mid$, regular for $alpha\_mid \leq \alpha < alpha\_max$ and interior for $alpha\_max \leq alpha$. The value *alpha_min* is undefined for faces which are not Gabriel faces and therefore do not appear in the alpha complex without any of their including face. The value *alpha_max* is undefined for convex hull faces which can never be interior, The data structure also includes two Boolean to mark if the face is a Gabriel face or a convex hull face.

The class *Alpha_status<NT>* is parameterized by a number type *NT*.

#include <CGAL/Alpha_shape_cell_base_3.h>

Creation

Alpha_status<NT> a;	default constructor.

Modifiers

void	*a.set_is_Gabriel(bool yesorno)*	
		sets Gabriel marker;
void	*a.set_is_on_chull(bool yesorno)*	
		sets convex hull marker;
void	*a.set_alpha_min(NT alpha)*	
		sets *alpha_min*.
void	*a.set_alpha_mid(NT alpha)*	
		sets *alpha_mid*.
void	*a.set_alpha_max(NT alpha)*	
		sets *alpha_max*.

Access Functions

bool	*a.is_Gabriel() const*	Returns true for Gabriel faces;
bool	*a.is_on_chull() const*	Returns true for convex hull faces;
NT	*a.alpha_min() const*	Returns the *alpha_min* *Precondition*: *is_Gabriel()* returns false;
NT	*a.alpha_mid() const*	Returns the *alpha_mid*;
NT	*a.alpha_max() const*	Returns *alpha_max*. *Precondition*: *is_on_chull()* returns false.

See Also

AlphaShapeCellBase_3
AlphaShapeVertexBase_3

Concept

WeightedAlphaShapeTraits_3

Definition

The concept WeightedAlphaShapeTraits_3 describes the requirements for the geometric traits class of the underlying regular triangulation of a weighted alpha shape.

Refines

RegularTriangulationTraits_3

In addition to the requirements described in the concept *RegularTriangulationTraits_3*, the geometric traits class of a Regular triangulation plugged in a basic alpha shapes provides the following.

Types

WeightedAlphaShapeTraits_3:: FT — A number type compatible with the type used for the points coordinates.

WeightedAlphaShapeTraits_3:: Compute_squared_radius_smallest_orthogonal_sphere_3

An object constructor able to compute the squared radius of the smallest sphere orthogonal to four weighted points *p0, p1, p2, p3*, and the squared radius of the smallest sphere orthogonal to three weighted points *p0, p1, p2*, and the squared radius of smallest sphere orthogonal to two weighted points *p0, p1*, and the squared radius of the smallest sphere orthogonal to a single point *p0*.

Creation

WeightedAlphaShapeTraits_3 wast; — default constructor.

Access Functions

Compute_squared_radius_smallest_orthogonal_sphere_3

wast.compute_squared_radius_smallest_orthogonal_sphere_3_object()

Has Models

CGAL::Regular_triangulation_euclidean_traits_3<K>

CGAL::Weighted_alpha_shape_euclidean_traits_3<K>

deprecated

Note : this class is deprecated since CGAL 3.6. *CGAL::Regular_triangulation_euclidean_traits_3* can be directly used instead.

Definition

The class *Weighted_alpha_shape_euclidean_traits_3<K>* is the default model for the concept *WeightedAlphaShapeTraits_3* of traits class for the underlying triangulation of a weighted alpha shapes. *K* must be a kernel.

#include <CGAL/Weighted_alpha_shape_euclidean_traits_3.h>

Refines

Regular_triangulation_euclidean_traits_3<K, typename K::FT>

Is Model for the Concepts

WeightedAlphaShapeTraits_3

deprecated

Class

Concept

FixedAlphaShapeTraits_3

Definition

The concept FixedAlphaShapeTraits_3 describes the requirements for the geometric traits class of the underlying Delaunay triangulation of a basic alpha shape with a fixed value alpha.

Refines

DelaunayTriangulationTraits_3

In addition to the requirements described in the concept *DelaunayTriangulationTraits_3*, the geometric traits class of a Delaunay triangulation plugged in a basic alpha shape with fixed alpha value provides the following.

Types

FixedAlphaShapeTraits_3:: Comparison_result

CGAL::Comparison_result or *Uncertain< CGAL::Comparison_result>*

FixedAlphaShapeTraits_3:: Compare_squared_radius_3

An object constructor able to compare
the squared radius of the smallest circumscribing sphere of either four, three, two or one point(s) with a given value of alpha. It provides:
Comparison_result operator()(Point_3,Point_3,Point_3,Point_3)
Comparison_result operator()(Point_3,Point_3,Point_3)
Comparison_result operator()(Point_3,Point_3)
Comparison_result operator()(Point_3)

Creation

FixedAlphaShapeTraits_3 traits; Default constructor.

Access Functions

Compare_squared_radius_3

traits.compare_squared_radius_3_object()

Has Models

All *CGAL* kernels.
CGAL::Exact_predicates_inexact_constructions_kernel (recommended)
CGAL::Exact_predicates_exact_constructions_kernel
CGAL::Filtered_kernel
CGAL::Cartesian
CGAL::Simple_cartesian
CGAL::Homogeneous
CGAL::Simple_homogeneous

Concept

FixedAlphaShapeCell_3

Definition

This concept describes the requirements for the base cell of a alpha shape with a fixed value alpha.

Refines

TriangulationCellBase_3.

Creation

FixedAlphaShapeCell_3 f;   default constructor.

FixedAlphaShapeCell_3 f(Vertex_handle v0, Vertex_handle v1, Vertex_handle v2, Vertex_handle v3);

constructor setting the incident vertices.

*FixedAlphaShapeCell_3 f(Vertex_handle v0,
Vertex_handle v1,
Vertex_handle v2,
Vertex_handle v3,
Cell_handle n0,
Cell_handle n1,
Cell_handle n2,
Cell_handle n3)*

constructor setting the incident vertices and the neighboring cells.

Access Functions

Classification_type   *f.get_classification_type()*

Returns the classification of the cell.

Classification_type   *f.get_facet_classification_type(int i)*

Returns the classification of the i-th facet of the cell.

Modifiers

void   *f.set_classification_type(Classification_type type)*

Sets classification of the cell.

void   *f.set_facet_classification_type(int i, Classification_type type)*

Sets the classification of the i-th facet of the cell.

FixedAlphaShapeVertex_3

Definition

This concept describes the requirements for the base vertex of a alpha shape with a fixed value alpha.

Refines

TriangulationVertexBase_3.

Types

FixedAlphaShapeVertex_3:: Point	Must be the same as the point type provided by the geometric traits class of the triangulation.

Creation

FixedAlphaShapeVertex_3	*v*;	default constructor.
FixedAlphaShapeVertex_3	*v(Point p);*	constructor setting the point associated to.
FixedAlphaShapeVertex_3	*v(Point p, Cell_handle c);*	constructor setting the point associated to and an incident cell.

Access Functions

bool	*v.is_on_chull()*	Returns a boolean indicating whether the point is on the convex hull of the point of the triangulation.
Classification_type	*v.get_classification_type()*	Returns the classification of the vertex.

Modifiers

void	*v.set_classification_type(Classification_type type)*	Sets the classification of the vertex.
void	*v.is_on_chull(bool b)*	Sets whether the vertex is on the convex hull.

Class

CGAL::Fixed_alpha_shape_3<Dt>

Definition

The class *Fixed_alpha_shape_3<Dt>* represents one (fixed) alpha shape of points in the 3D space for a real α. It maintains an underlying triangulation of the class *Dt* which represents connectivity and order among its faces. Each k-dimensional face of the *Dt* is associated with a classification that specifies its status in the alpha complex, alpha being fixed.

Note that this class can be used at the same time to build a *basic* or a *weighted* Alpha Shape.

#include <CGAL/Fixed_alpha_shape_3.h>

Inherits From

Dt

This class is the underlying triangulation class.

The modifying functions *insert* and *remove* will overwrite the inherited functions.

Types

Fixed_alpha_shape_3<Dt>:: Gt — the alpha shape traits type.

it has to derive from a triangulation traits class. For example *Dt::Point* is a Point class.

typedef Gt::FT

FT; — the number type of alpha.

enum Classification_type { EXTERIOR, SINGULAR, REGULAR, INTERIOR};

Enum to classify the simplices of the underlying triangulation with respect to a given alpha value.
Each k-dimensional simplex of the triangulation can be classified as EXTERIOR, SINGULAR, REGULAR or INTERIOR. A k simplex is REGULAR if it is on the boundary of the alpha complex and belongs to a $k+1$ simplex in this complex and it is SINGULAR if it is a boundary simplex that is not included in a $k+1$ simplex of the complex.

Creation

Fixed_alpha_shape_3<Dt> A(FT alpha = 0);

Introduces an empty alpha shape data structure *A* and set the alpha value to *alpha*.

Fixed_alpha_shape_3<Dt> A(Dt& dt, FT alpha = 0);

Builds an alpha shape from the triangulation *dt*, and set the alpha value to *alpha*. Be careful that this operation destroys the triangulation.

template < class InputIterator >
Fixed_alpha_shape_3<Dt> A(InputIterator first, InputIterator last, FT alpha = 0);

Builds an alpha shape data structure for the points in the range [*first*, *last*) and set the alpha value to *alpha*.
Requirement: The *value_type* of *first* and *last* is *Point* (the type point of the underlying triangulation.)

Modifiers

Vertex_handle

A.insert(Point p, Cell_handle start = Cell_handle())

Inserts point p in the underlying triangulation and returns the corresponding vertex. The optional argument *start* is used as a starting place for the search. The classification types of the new simplices are computed and that of the simplices incident to the new ones are updated.

void *A.remove(Vertex_handle v)*

Removes the vertex v from the underlying triangulation. The classification types of new simplices and their incident faces are set or reset

void *A.clear()* Clears the structure.

Query Functions

FT *A.get_alpha(void) const*

Returns the α-value.

Classification_type

A.classify(Cell_handle c) const

Classifies the cell *c* of the underlying triangulation in the alpha complex.

Classification_type

A.classify(Facet f) const

Classifies the facet *f* of the underlying triangulation in the alpha complex.

Classification_type

A.classify(Cell_handle f, int i) const

Classifies the facet of the cell *f* opposite to the vertex with index *i* of the underlying triangulation in the alpha complex.

Classification_type

A.classify(Edge e) const

Classifies the edge *e* of the underlying triangulation in the alpha complex.

Classification_type

A.classify(Vertex_handle v) const

Classifies the vertex *v* of the underlying triangulation in the alpha complex.

template<class OutputIterator>
OutputIterator

A.get_alpha_shape_cells(OutputIterator it, Classification_type type)

Writes the cells which are of type *type* in the alpha complex to the sequence pointed to by the output iterator *it*. Returns past the end of the output sequence.

template<class OutputIterator>
OutputIterator

A.get_alpha_shape_facets(OutputIterator it, Classification_type type)

Writes the facets which are of type *type* in the alpha complex to the sequence pointed to by the output iterator *it*. Returns past the end of the output sequence.

template<class OutputIterator>
OutputIterator

A.get_alpha_shape_edges(OutputIterator it, Classification_type type)

Writes the edges which are of type *type* in the alpha complex to the sequence pointed to by the output iterator *it*. Returns past the end of the output sequence.

template<class OutputIterator>

OutputIterator

A.get_alpha_shape_vertices(OutputIterator it, Classification_type type)

Writes the vertices which are of type *type* in the alpha complex to the sequence pointed to by the output iterator *it*. Returns past the end of the output sequence.

I/O

The I/O operators are defined for *iostream*, and for the window stream provided by CGAL. The format for the iostream is an internal format.

#*include* <*CGAL/IO/io.h*>

ostream&	*ostream& os* $<<$ *A*	Inserts the alpha shape *A* into the stream *os*. *Requirement*: The insert operator must be defined for *GT::Point*.

CGAL::Fixed_alpha_shape_cell_base_3<Traits,Fb>

Definition

The class *Fixed_alpha_shape_cell_base_3<Traits,Fb>* is the default model for the concept *FixedAlphaShapeCell_3*.

The class has two parameters. The traits class *Traits* provides the number type for alpha values. The second parameter *Fb* is a base class instantiated by default with *CGAL::Triangulation_cell_base_3<Traits>*.

#*include* <*CGAL/Fixed_alpha_shape_cell_base_3.h*>

Is Model for the Concepts

FixedAlphaShapeCellBase_3

Inherits From

Fb

CGAL::Fixed_alpha_shape_vertex_base_3<Traits,Vb>

Class

Definition

The class *Fixed_alpha_shape_vertex_base_3<Traits,Vb>* is the default model for the concept *FixedAlphaShapeVertex_3*.

The class has two parameters : the traits class *Traits* which provides the type for the points or the weighted points. The second parameter *Vb* is a base class instantiated by default with *CGAL::Triangulation_vertex_base_3<Traits>*.

#include <CGAL/Fixed_alpha_shape_vertex_base_3.h>

Is Model for the Concepts

FixedAlphaShapeVertexBase_3

Inherits From

Vb

Concept

FixedWeightedAlphaShapeTraits_3

Definition

The concept FixedWeightedAlphaShapeTraits_3 describes the requirements for the geometric traits class of the underlying regular triangulation of a weighted alpha shape with fixed alpha value.

Refines

RegularTriangulationTraits_3

In addition to the requirements described in the concept *RegularTriangulationTraits_3*, the geometric traits class of a Regular triangulation plugged in a weighted alpha shape with fixed alpha value provides the following.

Types

FixedWeightedAlphaShapeTraits_3:: Comparison_result

CGAL::Comparison_result or *Uncertain< CGAL::Comparison_result>*

FixedWeightedAlphaShapeTraits_3:: Compare_weighted_squared_radius_3

An object constructor able to compare the squared radius of the smallest sphere orthogonal to either four, three, two or one weighted point(s) to a given value of alpha. It provides:
Comparison_result operator()(Weighted_point_3 ,Weighted_point_3 ,Weighted_point_3 ,Weighted_point_3)
Comparison_result operator()(Weighted_point_3 ,Weighted_point_3 ,Weighted_point_3)
Comparison_result operator()(Weighted_point_3 ,Weighted_point_3)
Comparison_result operator()(Weighted_point_3)

Creation

FixedWeightedAlphaShapeTraits_3 wast; default constructor.

Access Functions

Compare_weighted_squared_radius_3

wast.compare_weighted_squared_radius_3_object()

Has Models

CGAL::Regular_triangulation_euclidean_traits_3<K>,

Part X

Voronoi Diagrams

Chapter 44

2D Segment Delaunay Graphs

Menelaos Karavelas

Contents

This chapter describes the two-dimensional segment Delaunay graph package of CGAL. We start with a few definitions in Section 44.1. The software design of the 2D segment Delaunay graph package is described in Section 44.2. In Section 44.3 we discuss the geometric traits of the 2D segment Delaunay graph package and in Section 44.4 the segment Delaunay graph hierarchy, a data structure suitable for fast nearest neighbor queries, is briefly described.

44.1 Definitions

The 2D segment Delaunay graph package of CGAL is designed to compute the Delaunay graph of a set of possibly intersecting segments on the plane. Although we compute the Delaunay graph, we will often refer to its dual, the segment Voronoi diagram, since it is easier to explain and understand. The algorithm that has been implemented is incremental. The corresponding CGAL class is called *Segment_Delaunay_graph_2< SegmentDelaunayGraphTraits_2,SegmentDelaunayGraphStructure_2>* and will be discussed in more detail in the sequel. The interested reader may want to refer to the paper by Karavelas [Kar04] for the general idea as well as the details of the algorithm implemented.

Definitions. Before describing the details of the implementation we make a brief introduction to the theory of segment Delaunay graphs and segment Voronoi diagrams. The segment Voronoi diagram is defined over a set

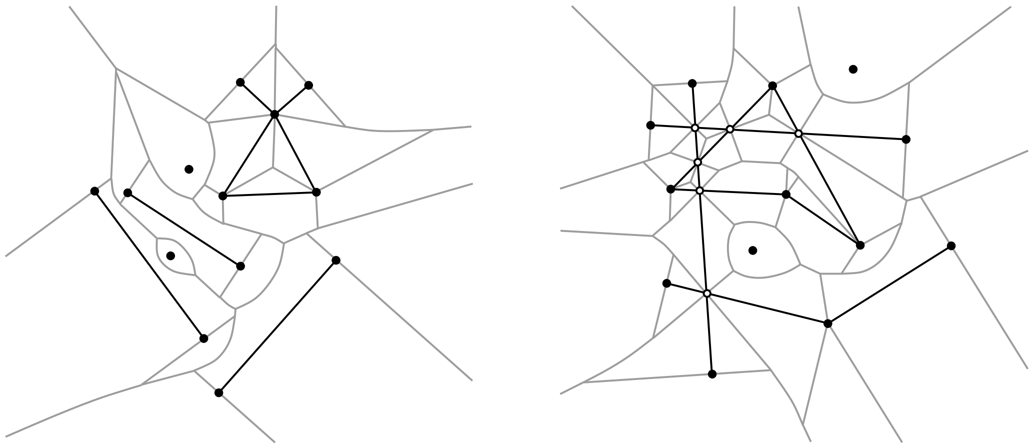


Figure 44.1: The segment Voronoi diagram for a set of weakly (left) and strongly (right) intersecting sites.

of non-intersecting sites, which can either be points or linear segments, which we assume that are given through their endpoints. The segment Voronoi diagram a subdivision of the plane into connected regions, called *cells*, associated with the sites. The dual graph of the segment Voronoi diagram is called the segment Delaunay graph. The cell of a site t_i is the locus of points on the plane that are closer to t_i than any other site t_j, $j \neq i$. The distance $\delta(x,t_i)$ of a point x in the plane to a site t_i is defined as the minimum of the Euclidean distances of x from the points in t_i. Hence, if t_i is a point p_i, then

$$\delta(x,t_i) = \|x - t_i\|,$$

whereas if t_i, is a segment, then

$$\delta(x,t_i) = \min_{y \in t_i} \|x - y\|,$$

where $\|\cdot\|$ denotes the Euclidean norm. It can easily be seen that it is a generalization of the Voronoi diagram for points.

In many applications the restriction that sites are non-intersecting is too strict. Often we want to allow segments that touch at their endpoints, or even segments that overlap or intersect properly at their interior (for example, see Fig. 44.1). Allowing such configurations poses certain problems. More specifically, when we allow segments to touch at their endpoints we may end up with pairs of segments whose bisector is two-dimensional. If we allow pairs of segments that intersect properly at their interior, the interiors of their Voronoi cells are no longer simply connected. In both cases above the resulting Voronoi diagrams are no longer instances of abstract Voronoi diagrams (cf. [Kle89]), which has a direct consequence on the efficient computation of the corresponding Voronoi diagram. The remedy to these problems is to consider linear segments not as one object, but rather as three, namely the two endpoints and the interior. This choice guarantees that all bisectors in the Voronoi diagram are one-dimensional and that all Voronoi cells are simply connected. Moreover, we further distinguish between two cases, according to the type of intersecting pair that our input data set contains. A pair of sites is called *weakly intersecting* if they have a single common point and this common point does not lie in the interior of any of the two sites. A pair of sites is called *strongly intersecting* if they intersect and they either have more than one common point or their common point lies in the interior of at least one of the two sites. As it will be seen later the two cases have different representation (and thus storage) requirements, as well as they require a somehow different treatment on how the predicates are evaluated. Having made the distinction between weakly and strongly intersecting sites, and having said that segment sites are treated as three objects, we are now ready to precisely define the Delaunay graph we compute. Given a set $\mathcal{S}$ of input sites, let $\mathcal{S}_{\mathcal{A}}$ be the set of points and (open) segments in the arrangement $\mathcal{A}(\mathcal{S})$ of $\mathcal{S}$. The 2D segment Delaunay graph package of CGAL computes the (triangulated) Delaunay graph that is dual to the Euclidean Voronoi diagram of the sites in the set $\mathcal{S}_{\mathcal{A}}$.

The segment Delaunay graph is uniquely defined once we have the segment Voronoi diagram. If the all sites are in *general position*, then Delaunay graph is a graph with triangular faces away from the convex hull of the set of sites. To unify our approach and handling of the Delaunay graph we add to the set of (finite) sites a fictitious site at infinity, which we call the *site at infinity*. We can then connect all vertices of the outer face of the Delaunay graph to the site at infinity which gives us a graph with the property that all of its faces are now triangular. However, the Delaunay graph is not a triangulation for two main reasons: we cannot always embed it on the plane with straight line segments that yield a triangulation and, moreover, we may have two faces of the graph that have two edges in common, which is not allowed in a triangulation.

We would like to finish our brief introduction to the theory of segment Delaunay graphs and segment Voronoi diagrams by discussing the concept of general position. We say that a set of sites is in general position if no two triplets of sites have the same tritangent Voronoi circle. This statement is rather technical and it is best understood in the context of points. The equivalent statement for points is that we have no two triplets of points that define the same circumcircle, or equivalently that no four points are co-circular. The statement about general position made above is a direct generalization of the (much simpler to understand) statement about points. On the contrary, when we have sites in degenerate position, the Delaunay graph has faces with more than three edges on their boundary. We can get a triangulated version of the Delaunay graph by simply *triangulating* the corresponding faces in an arbitrary way. In fact the algorithm that has been implemented in CGAL has the property that it always returns a valid *triangulated* version of the segment Delaunay graph. By valid we mean that it contains the actual (non-triangulated) Delaunay graph, and whenever there are faces with more than three faces then they are triangulated. The way that they are triangulated depends on the order of insertion of the sites in the diagram.

One final remark has to be made with respect to the difference between the set of *input sites* and the set of *output sites*. The set of input sites consists of the closed sites that the user inserts in the diagram. Since segment sites are treated as three objects, internally our algorithm sees only points and open segments. As a result, from the point of view of the algorithm, the input sites have no real meaning. What has real meaning is the set of sites that correspond to cells of the Voronoi diagram and this is the set of output sites.

Degenerate Dimensions. The dimension of the segment Delaunay graph is in general 2. The exceptions to this rule are as follows:

- The dimension is -1 if the segment Delaunay graph contains no sites.
- The dimension is 0 if the segment Delaunay graph contains exactly one (output) site.
- The dimension is 1 is the segment Delaunay graph contains exactly two (output) sites.

44.2 Software Design

The 2D segment Delaunay graph class *Segment_Delaunay_graph_2<SegmentDelaunayGraphTraits_2,SegmentDelaunayGraphDataStructure_2>* follows the design of the triangulation package of CGAL. It is parametrized by two arguments:

- the **geometric traits** class. It provides the basic geometric objects involved in the algorithm, such as sites, points etc. It also provides the geometric predicates for the computation of the segment Delaunay graph, as well as some basic constructions that can be used, for example, to visualize the diagram. The geometric traits for the segment Delaunay graph will be discussed in more detail in the next section.
- the **segment Delaunay graph data structure**. This is essentially the same as the Apollonius graph data structure (discussed in Chapter 45.2), augmented with some additional operations that are specific to segment Voronoi diagrams. The corresponding concept is that of *SegmentDelaunayGraphDataStructure_2*,

which in fact is a refinement of the *ApolloniusGraphDataStructure_2* concept. The class *Triangulation_data_structure_2<Vb,Fb>* is a model of the concept *SegmentDelaunayGraphDataStructure_2*. A default value for the corresponding template parameter is provided, so the user does not need to specify it.

Strongly Intersecting Sites and their Representation. As we have mentioned above, the segment Delaunay graph package of CGAL is designed to support the computation of the segment Voronoi diagram even when the input segment sites are intersecting. This choice poses certain issues for the design of the software package. The major concern is the representation of the subsegments that appear in the arrangement of the these sites, because the sites in the arrangement are the ones over which the diagram is actually defined. A direct consequence of the choice of representation is the algebraic degree of the predicates involved in the computation of the segment Delaunay graph, as well as the storage requirements for the subsegments and points on intersection in the arrangement.

The case of weakly intersecting sites does not require any special treatment. We can simply represent points by their coordinates and segments by their endpoints. In the case of strongly intersecting sites, the obvious choice to use the afore-mentioned representation has severe disadvantages. Consider two strongly intersecting segments t_i and t_j, whose endpoints have homogeneous coordinates of size b. Their intersection point will have homogeneous coordinates of bit size $6b+O(1)$. This effect can be cascaded, which implies that after inserting k (input) segments we can arrive at having points of intersection whose bit sizes are exponential with respect to k, i.e., their homogeneous coordinates will have bit size $\Omega(2^k b)$. Not only the points of intersection, but also the adjacent subsegments will be represented by quantities of arbitrarily high bit size, and as a result we would not be able to give a bound on the bit sizes of the coordinates of the points of intersection. As a result, we would not be able to give a bound on the memory needed to store these coordinates. An equally important consequence is that we would also not be able to give a bound on the algebraic degree of the algebraic expressions involved in the evaluation of the predicates.

Such a behavior is obviously undesirable. For robustness, efficiency, and scalability purposes, it is critical that the bit size of the algebraic expressions in the predicates does not depend on the input size. For this reason, as well as for others to be discussed below, we decided to represent sites in a implicit manner, which somehow encodes the history of their construction. In particular, we exploit the fact that points of intersection always lie on two input segments, and that segments that are not part of the input are always supported by input segments.

For example, let us consider the configuration in Fig. 44.2. We assume that the segments $t_i = p_i q_i$, $i = 1,2,3$, are inserted in that order. Upon the insertion of t_2, our algorithm will split the segment t_1 into the subsegments $p_1 s_1$ and $s_1 q_1$, then add s_1, and finally insert the subsegments $p_2 s_1$ and $s_1 q_2$. How do we represent the five new sites? s_1 will be represented by its two defining segments t_1 and t_2. The segment $p_1 s_1$ will be represented by two segments, a point, and a boolean. The first segment is t_1, which is always the segment with the same support as the newly created segment. The second segment is t_2 and the point is p_1. The boolean indicates whether the first endpoint of $p_1 s_1$ is an input point; in this case the boolean is equal to *true*. The segment $s_1 q_1$ will also be represented by two segments, a point, and a boolean, namely, t_1 (the supporting segment of $s_1 q_1$), t_2 and *false* (it is the second endpoint of $s_1 q_1$ that is an input point). Subsegments $p_2 s_1$ and $s_1 q_2$ are represented analogously. Consider now what happens when we insert t_3. The point s_2 will again be represented by two segments, but not $s_1 q_1$ and t_3. In fact, it will be represented by t_1 (the supporting segment of $s_1 q_1$) and t_3. $s_2 q_1$ will be represented by two segments, a point, and a boolean (t_1, t_3, $q1$ and *false*), and similarly for $p_3 s_2$ and $s_2 q_3$. On the other hand, both endpoints of $s_1 s_2$ are non-input points. In such a case we represent the segment by three input segments. More precisely, $s_1 s_2$ is represented by the segments t_1 (the supporting segment of $s_1 q_1$), t_2 (it defines s_1 along with t_1) and t_3 (it defines s_2 along with t_1).

The five different presentations, two for points (coordinates; two input segments) and three for segments (two input points; two input segments, an input point and a boolean; three input segments), form a closed set of representations and thus represent any point of intersection or subsegment regardless of the number of input segments. Moreover, every point (input or intersection) has homogeneous coordinates of bit size at most $3b+O(1)$. The supporting lines of the segments (they are needed in some of the predicates) have coefficients which are always of bit size $2b+O(1)$. As a result, the bit size of the expressions involved in our predicates will always

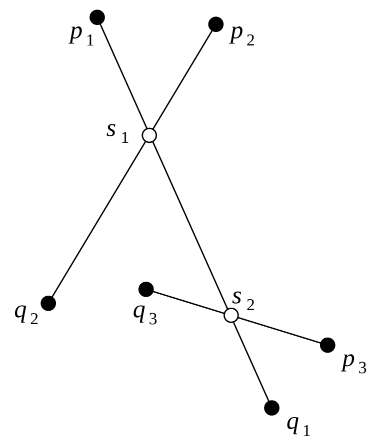


Figure 44.2: Site representation. The point s_1 is represented by the four points p_1, q_1, p_2 and q_2. The segment $p_1 s_1$ is represented by the points p_1, q_1, p_2, q_2 and a boolean which is set to *true* to indicate that the first endpoint in not a point of intersection. The segment $s_1 s_2$ is represented by the six points: p_1, q_1, p_2, q_2, p_3 and q_3. The remaining (non-input) points and segments in the figure are represented similarly.

be $O(b)$, independently of the size of the input. The *SegmentDelaunayGraphSite_2* concept encapsulates the ideas presented above. A site is represented in this concept by up to four points and a boolean, or up to six points, depending on its type. The class *Segment_Delaunay_graph_site_2<K>* implements this concept.

Even this representation, however, has some degree of redundancy. The endpoint of a segment appears in both the representation of the (open) segment site as well as the representation of the point site itself. The situation becomes even worse in the presence of strongly intersecting sites: a point may appear in the representation of multiple subsegments and/or points of intersection. To avoid this redundancy, input points are stored in a container, and the various types of sites (input points and segments, points of intersection, subsegments with one or two points of intersection as endpoints) only store handles to the points in the container. This is achieved by the *Segment_Delaunay_graph_storage_site_2<Gt>* class which is a model of the corresponding concept: *SegmentDelaunayGraphStorageSite_2*. This concept enforces a site to be represented by up to 6 handles (which are very lightweight objects) instead of 6 points, which are, compared to handles of course, very heavy objects.

Optimizing Memory Allocation. There are applications where we know beforehand that the input consists of only weakly intersecting sites. In these cases the site representation described above poses a significant overhead in the memory requirements of our implementation: instead of representing sites with up to two points (or ultimately with to two handles), we require sites to store six points (respectively, six handles). To avoid this overhead we have introduced two series of traits classes:

- One that supports the full-fledged sites, and is suitable when the input consists of strongly intersecting sites. This series consists of the *Segment_Delaunay_graph_traits_2<K,MTag>* and *Segment_Delaunay_graph_filtered_traits_2<CK,CM,EK,EM,FK,FM>* classes.
- One that is customized for input that contain only weakly intersecting sites. This series consists of the *Segment_Delaunay_graph_traits_without_intersections_2<K,MTag>* and *Segment_Delaunay_graph_filtered_traits_without_intersections_2<CK,CM,EK,EM,FK,FM>* classes.

The advantages of having different traits classes are as follows:

- When the user chooses to use one of the traits classes in the second series we only store two handles per site. This implies a reduction by a factor of three in the memory allocated per site with respect to the first series of traits classes.

- In the case of the first series of traits classes, we can better exploit the knowledge that have strongly intersecting sites, in order to further apply geometric filters (see below) during the evaluation of the predicates. On the contrary, if the second series of traits classes is used, we can avoid geometric filtering tests that have meaning only in the case of strongly intersecting sites.

44.3 The Geometric Traits

The predicates required for the computation of the segment Voronoi diagram are rather complicated. It is not the purpose of this document to discuss them in detail. The interested reader may refer to Burnikel's thesis [Bur96], where it is shown that in the case of weakly intersecting sites represented in homogeneous coordinates of bit size b, the maximum bit size of the algebraic expressions involved in the predicates is $40b+O(1)$. Given our site representation given above we can guarantee that even in the case of strongly intersecting sites, the algebraic degree of the predicates remains $O(b)$, independently of the size of the input. What we want to focus in the remainder of this section are the different kinds of filtering techniques that we have employed in our implementation.

Geometric Filtering. Our representation of sites is coupled very naturally, with what we call *geometric filtering*. The technique amounts to performing simple geometric tests exploiting the representation of our data, as well as the geometric structure inherent in our problem, in order to evaluate predicates in seemingly degenerate configurations. Geometric filtering can be seen as a preprocessing step before performing arithmetic filtering. Roughly speaking, by arithmetic filtering we mean that we first try to evaluate the predicates using a fixed-precision floating-point number type (such as `double`), and at the same time keep error bounds on the numerical errors of the computations we perform. If the numerical errors are too big and do not permit us to evaluate the predicate, we switch to an exact number type, and repeat the evaluation of the predicate. Geometric filtering can help by eliminating situations in which the arithmetic filter will fail, thus decreasing the number of times we need to evaluate a predicate using exact arithmetic.

To illustrate the application and effectiveness of this approach, let us consider a very simple example usage. Suppose we want to determine if two non-input points are identical (we assume here that the input sites are represented by *double*s). In order to do that we need to compute their coordinates and compare them. If the two points are identical, the answer to our question using *double* arithmetic may be wrong (due to numerical errors), in which case we will have to reside to the more expensive exact computation. Instead, before testing the coordinates for equality, we can use the representation of the points to potentially answer the question. More specifically, and this is the geometric filtering part of the computation, we can first test if the defining segments of the two points are the same. If they are not, then we proceed to comparing their coordinates as usual. Testing the defining segments for equality does not involve any arithmetic operations on the input, but rather only comparisons on *double*s. By performing this very simple test we avoid a numerically difficult computation, which could be performed thousands of times during the computation of a Delaunay graph.

Geometric filtering has been implemented in all our models of the *SegmentDelaunayGraphTraits_2* concept. These models are the classes: *Segment_Delaunay_graph_traits_2<K,MTag>*, *Segment_Delaunay_graph_traits_without_intersections_2<K,MTag>*, *Segment_Delaunay_graph_filtered_traits_2<CK,CM,EK,EM,FK,FM>* and *Segment_Delaunay_graph_filtered_traits_without_intersections_2<CK,CM,EK,EM,FK,FM>*.

Arithmetic Filtering. As mentioned above, performing computations with exact arithmetic can be very costly. For this reason we have devoted considerable effort in implementing different kinds of arithmetic filtering mechanisms. Presently, there two ways of performing arithmetic filtering for the predicates involved in the computation of segment Delaunay graphs:

1. The user can define his/her kernel using as number type, a number type of the form *CGAL::Filtered_exact<CT,ET>*. Then this kernel can be entered as the first template parameter in the *Segment_Delaunay_graph_2<K,MTag>* or *Segment_Delaunay_graph_without_intersections_2<K,MTag>* class.

2. The user can define up to three different kernels *CK*, *FK* and *EK* (default values are provided for most parameters). The first kernel *CK* is used only for constructions. The second kernel *FK* is the filtering kernel: the traits class will attempt to compute the predicates using this kernel. If the filtering kernel fails to successfully compute a predicate, the exact kernel *EK* will be used. These three kernels are then used in the *Segment_Delaunay_graph_filtered_traits_2<CK,CM,EK,EM,FK,FM>* and *Segment_Delaunay_graph_filtered_traits_without_intersections_2<CK,CM,EK,EM,FK,FM>* classes, which have been implemented using the *Filtered_predicate<EP,FP>* mechanism.

Our experience so far has shown that for all reasonable and valid values of the template parameters, the second method for arithmetic filtering is more efficient among the two.

Let's consider once more the classes *Segment_Delaunay_graph_2<K,MTag>* and *Segment_Delaunay_graph_without_intersections_2<K,MTag>*. The template parameter *MTag* provides another degree of freedom to the user, who can indicate the type of arithmetic operations to be used in the evaluation of the predicates. More specifically, in both classes, *MTag* can be *CGAL::Field_with_sqrt_tag*, in which case the predicates will be evaluated using all four basic arithmetic operations plus square roots; this requires, of course, that the number type used in the kernel *K* supports these operations exactly. The second choices are *CGAL::Field_tag* for the *Segment_Delaunay_graph_2<K,MTag>* class, and *CGAL::Euclidean_ring_tag* for the *Segment_Delaunay_graph_without_intersections_2<K,MTag>* class. In the first case we indicate that we want the predicates to be computed using only the four basic arithmetic operations, whereas in the second case we evaluate the predicates using only ring operations. Again, for the predicates to be evaluated correctly, the number type used in the kernel *K* must support the corresponding operations exactly.

The semantics for the template parameters *CM*, *FM* and *EM* in the *Segment_Delaunay_graph_filtered_traits_2<CK,CM,EK,EM,FK,FM>* and *Segment_Delaunay_graph_filtered_traits_without_intersections_2<CK,CM,EK,EM,FK,FM>* classes are analogous. With each of these template parameters we can control the type of arithmetic operations that are going to be used in calculations involving each of the corresponding kernels *CK*, *FK* and *EK*. When the *Segment_Delaunay_graph_filtered_traits_2<CK,CM,EK,EM,FK,FM>* is used the possible values for *CM*, *FM* and *EM* are *CGAL::Field_with_sqrt_tag* and *CGAL::Field_tag*, whereas if the *Segment_Delaunay_graph_filtered_traits_without_intersections_2<CK,CM,EK,EM,FK,FM>* class is used, the possible values are *CGAL::Field_with_sqrt_tag* and *CGAL::Euclidean_ring_tag*. The semantics are the same as in the case of the *Segment_Delaunay_graph_2<K,MTag>* and *Segment_Delaunay_graph_without_intersections_2<K,MTag>* classes.

44.4 The Segment Delaunay Graph Hierarchy

The *Segment_Delaunay_graph_hierarchy_2<SegmentDelaunayGraphTraits_2, SSTag, SegmentDelaunayGraphDataStructure_2>* class is the analogue of the *Triangulation_hierarchy_2* or the *Apollonius_graph_hierarchy_2* classes, applied to the segment Delaunay graph. It consists of a hierarchy of segment Delaunay graphs constructed in a manner analogous to the Delaunay hierarchy by Devillers [Dev02]. Unlike the triangulation hierarchy or the Apollonius graph hierarchy, the situation here is more complicated because of two factors: firstly, segments are treated as three objects instead of one (the two endpoints and the interior of the segments), and secondly, the presence of strongly intersecting sites complicates significantly the way the hierarchy is constructed. The interested reader may refer to the paper by Karavelas [Kar04] for the details of the construction of the hierarchy. Another alternative is to have a hybrid hierarchy that consists of the segment Delaunay graph at the bottom-most level and point Voronoi diagrams at all other levels. This choice seems to work very well in practice , primarily because it avoids the overhead of maintaining a Delaunay graph for segments at the upper levels of the hierarchy. However, it seems much less likely to be possible to give any theoretical guarantees for its performance, in contrast to the hierarchy with segment Delaunay graphs at all levels (cf. [Kar04]). The user can choose between the two types of hierarchies by means of the template parameter *SSTag*. If *SSTag* is set to *false* (which is also the default value), the upper levels of the hierarchy consist of point Delaunay graphs. If *SSTag* is set to *true*, we have segment Delaunay graphs at all levels of the hierarchy.

The class *Segment_Delaunay_graph_hierarchy_2<SegmentDelaunayGraphTraits_2, SSTag, SegmentDelaunayGraphDataStructure_2>* has exactly the same interface and functionality as the *Segment_Delaunay_graph_2<SegmentDelaunayGraphTraits_2,SegmentDelaunayGraphDataStructure_2>* class. Using the segment Delaunay graph hierarchy involves an additional cost in space and time for maintaining the hierarchy. Our experiments have shown that it usually pays off to use the hierarchy for inputs consisting of more than about 1,000 sites.

44.5 Examples

44.5.1 First Example

The following example shows how to use the segment Delaunay graph traits in conjunction with the *Filtered_exact<CT,ET>* mechanism. In addition it shows how to use a few of the iterators provided by the *Segment_Delaunay_graph_2* class in order to count a few site-related quantities.

```
#include <iostream>
#include <fstream>
#include <cassert>

// define the exact number type
#  include <CGAL/Quotient.h>
#  include <CGAL/MP_Float.h>
typedef CGAL::Quotient<CGAL::MP_Float>  ENT;

// define the kernels
#include <CGAL/Simple_cartesian.h>

typedef CGAL::Simple_cartesian<double>     CK;
typedef CGAL::Simple_cartesian<ENT>        EK;

// typedefs for the traits and the algorithm
#include <CGAL/Segment_Delaunay_graph_filtered_traits_2.h>
#include <CGAL/Segment_Delaunay_graph_2.h>

typedef CGAL::Segment_Delaunay_graph_filtered_traits_2<CK,
/* The construction kernel allows for / and sqrt */
   CGAL::Field_with_sqrt_tag,
                                                        EK,
/* The exact kernel supports field ops exactly */      CGAL::Field_tag>
   Gt;

typedef CGAL::Segment_Delaunay_graph_2<Gt>             SDG2;
```

```
using namespace std;

int main() {
  ifstream ifs("data/sitesx.cin");
  assert( ifs );

  SDG2          sdg;
  SDG2::Site_2  site;

  while ( ifs >> site ) { sdg.insert( site ); }

  ifs.close();

  assert( sdg.is_valid(true, 1) );
  cout << endl << endl;

  // print the number of input and output sites
  cout << "# of input sites : " << sdg.number_of_input_sites() << endl;
  cout << "# of output sites: " << sdg.number_of_output_sites() << endl;

  unsigned int n_ipt(0), n_iseg(0), n_opt(0), n_oseg(0), n_ptx(0);

  // count the number of input points and input segments
  SDG2::Input_sites_iterator iit;
  for (iit = sdg.input_sites_begin(); iit != sdg.input_sites_end();
   ++iit)
    {
      if ( iit->is_point() ) { n_ipt++; } else { n_iseg++; }
    }

  // count the number of output points and output segments, as well
  // as the number of points that are points of intersection of pairs
  // of strongly intersecting sites
  SDG2::Output_sites_iterator oit;
  for (oit = sdg.output_sites_begin(); oit != sdg.output_sites_end();
   ++oit)
    {
      if ( oit->is_segment() ) { n_oseg++; } else {
        n_opt++;
        if ( !oit->is_input() ) { n_ptx++; }
      }
    }

  cout << endl << "# of input segments:  " << n_iseg << endl;
  cout << "# of input points:    " << n_ipt << endl << endl;
  cout << "# of output segments: " << n_oseg << endl;
  cout << "# of output points:   " << n_opt << endl << endl;
  cout << "# of intersection points: " << n_ptx << endl;

  return 0;
}
```

File: examples/Segment_Delaunay_graph_2/sdg-count-sites.cpp

44.5.2 Second Example

The following example shows how to use the segment Delaunay graph hierarchy along with the filtered traits class that supports intersecting sites.

```
#include <iostream>
#include <fstream>
#include <cassert>

// example that uses the filtered traits and
// the segment Delaunay graph hierarchy

// choose the kernel
#include <CGAL/Simple_cartesian.h>

struct Rep : public CGAL::Simple_cartesian<double> {};

// typedefs for the traits and the algorithm
#include <CGAL/Segment_Delaunay_graph_hierarchy_2.h>
#include <CGAL/Segment_Delaunay_graph_filtered_traits_2.h>

struct Gt
  : public CGAL::Segment_Delaunay_graph_filtered_traits_2<Rep> {};

typedef CGAL::Segment_Delaunay_graph_hierarchy_2<Gt>  SDG2;


int main()
{
  std::ifstream ifs("data/sites.cin");
  assert( ifs );

  SDG2          sdg;
  SDG2::Site_2  site;

  // read the sites and insert them in the segment Delaunay graph
  while ( ifs >> site ) {
    sdg.insert(site);
  }

  // validate the segment Delaunay graph
  assert( sdg.is_valid(true, 1) );

  return 0;
}
```

File: examples/Segment_Delaunay_graph_2/sdg-filtered-traits.cpp

44.5.3 Third Example

The following example demonstrates how to recover the defining sites for the edges of the Voronoi diagram (which are the duals of the edges of the segment Delaunay graph computed).

```
#include <iostream>
#include <fstream>
#include <cassert>
#include <string>

// define the kernel
#include <CGAL/Simple_cartesian.h>
#include <CGAL/Filtered_kernel.h>

typedef CGAL::Simple_cartesian<double>    CK;
typedef CGAL::Filtered_kernel<CK>         Kernel;

// typedefs for the traits and the algorithm
#include <CGAL/Segment_Delaunay_graph_traits_2.h>
#include <CGAL/Segment_Delaunay_graph_2.h>

typedef CGAL::Segment_Delaunay_graph_traits_2<Kernel>  Gt;
typedef CGAL::Segment_Delaunay_graph_2<Gt>             SDG2;

using namespace std;

int main()
{
  ifstream ifs("data/sites2.cin");
  assert( ifs );

  SDG2          sdg;
  SDG2::Site_2  site;

  // read the sites from the stream and insert them in the diagram
  while ( ifs >> site ) { sdg.insert( site ); }

  ifs.close();

  // validate the diagram
  assert( sdg.is_valid(true, 1) );
  cout << endl << endl;

  /*
  // now walk through the non-infinite edges of the segment Delaunay
  // graphs (which are dual to the edges in the Voronoi diagram) and
  // print the sites defining each Voronoi edge.
  //
  // Each oriented Voronoi edge (horizontal segment in the figure
  // below) is defined by four sites A, B, C and D.
  //
  //       \                     /
  //        \         B         /
  //         \                 /
```

```
  //     C  -----------------  D
  //       /                  \
  //      /         A          \
  //     /                      \
  //
  // The sites A and B define the (oriented) bisector on which the
  // edge lies whereas the sites C and D, along with A and B define
  // the two endpoints of the edge. These endpoints are the Voronoi
  // vertices of the triples A, B, C and B, A, D.
  // If one of these vertices is the vertex at infinity the string
  // "infinite vertex" is printed; the corresponding Voronoi edge is
  // actually a stright-line or parabolic ray.
  // The sites below are printed in the order A, B, C, D.
  */

  string inf_vertex("infinite vertex");
  char vid[] = {'A', 'B', 'C', 'D'};

  SDG2::Finite_edges_iterator eit = sdg.finite_edges_begin();
  for (int k = 1; eit != sdg.finite_edges_end(); ++eit, ++k) {
    SDG2::Edge e = *eit;
    // get the vertices defining the Voronoi edge
    SDG2::Vertex_handle v[] = { e.first->vertex( sdg.ccw(e.second) ),
                                e.first->vertex( sdg.cw(e.second) ),
                                e.first->vertex( e.second ),
                                sdg.tds().mirror_vertex(e.first,
   e.second) };

    cout << "--- Edge " << k << " ---" << endl;
    for (int i = 0; i < 4; i++) {
      // check if the vertex is the vertex at infinity; if yes, print
      // the corresponding string, otherwise print the site
      if ( sdg.is_infinite(v[i]) ) {
        cout << vid[i] << ": " << inf_vertex << endl;
      } else {
        cout << vid[i] << ": " << v[i]->site() << endl;
      }
    }
    cout << endl;
  }

  return 0;
}
```

File: examples/Segment_Delaunay_graph_2/sdg-voronoi-edges.cpp

2D Segment Delaunay Graphs Reference Manual

Menelaos Karavelas

CGAL provides the class *CGAL::Segment_Delaunay_graph_2<Gt,DS>* for computing the 2D Delaunay graph of segments and points. The two template parameters must be models of the *SegmentDelaunayGraphTraits_2* and *SegmentDelaunayGraphDataStructure_2* concepts. The first concept is related to the geometric objects and predicates associated with segment Delaunay graphs, whereas the second concept refers to the data structure used to represent the segment Delaunay graph, which is dual to the 2D Voronoi diagram of segments and points. The classes *Segment_Delaunay_graph_traits_2<K,MTag>*, *Segment_Delaunay_graph_traits_without_intersections_2<K,MTag>* *Segment_Delaunay_graph_filtered_traits_2<K,MTag>* *Segment_Delaunay_graph_filtered_traits_without_intersections_2<K,MTag>* are models of the *SegmentDelaunayGraphTraits_2* concept, whereas the class *Triangulation_data_structure_2<Vb,Fb>* is a model of the *SegmentDelaunayGraphDataStructure_2* concept.

44.6 Classified Reference Pages

Concepts

Classes

44.7 Alphabetical List of Reference Pages

CGAL::Segment_Delaunay_graph_2<Gt,DS>

Definition

The class *Segment_Delaunay_graph_2<Gt,DS>* represents the segment Delaunay graph (which is the dual graph of the 2D segment Voronoi diagram). Currently it supports only insertions of sites. It is templated by two template arguments *Gt*, which must be a model of *SegmentDelaunayGraphTraits_2* and *DS*, which must be a model of *SegmentDelaunayGraphDataStructure_2*. The second template argument defaults to *CGAL::Triangulation_data_structure_2< CGAL::Segment_Delaunay_graph_vertex_base_2<Gt>, CGAL::Triangulation_face_base_2< Gt> >*.

#include <CGAL/Segment_Delaunay_graph_2.h>

Is Model for the Concepts

DelaunayGraph_2

Types

typedef Gt	*Geom_traits;*	A type for the geometric traits.
typedef DS	*Data_structure;*	A type for the underlying data structure.
typedef Data_structure	*Triangulation_data_structure;*	
		This type has been added so that the *Segment_Delaunay_graph_2<Gt,DS>* class is a model of the *DelaunayGraph_2* concept.
typedef typename DS::size_type	*size_type;*	Size type (an unsigned integral type)
typedef typename Gt::Point_2	*Point_2;*	A type for the point defined in the geometric traits.
typedef typename Gt::Site_2	*Site_2;*	A type for the segment Delaunay graph site, defined in the geometric traits.
Segment_Delaunay_graph_2<Gt,DS>:: Point_container		A type for the container of points.
typedef typename Point_container::iterator	*Point_handle;*	A handle for points in the point container.

The vertices and faces of the segment Delaunay graph are accessed through *handles*, *iterators* and *circulators*. The iterators and circulators are all bidirectional and non-mutable. The circulators and iterators are assignable to the corresponding handle types, and they are also convertible to the corresponding handles. The edges of the segment Delaunay graph can also be visited through iterators and circulators, the edge circulators and iterators are also bidirectional and non-mutable. In the following, we call *infinite* any face or edge incident to the infinite vertex and the infinite vertex itself. Any other feature (face, edge or vertex) of the segment Delaunay graph is said to be *finite*. Some iterators (the *All* iterators) allow to visit finite or infinite features while the others (the *Finite* iterators) visit only finite features. Circulators visit both infinite and finite features.

typedef typename DS::Edge	*Edge;*	The edge type. The *Edge(f,i)* is the edge common to faces *f* and *f.neighbor(i)*. It is also the edge joining the vertices *vertex(cw(i))* and *vertex(ccw(i))* of *f*. *Precondition*: *i* must be *0*, *1* or *2*.

typedef typename DS::Vertex	*Vertex;*	A type for a vertex.
typedef typename DS::Face	*Face;*	A type for a face.
typedef typename DS::Vertex_handle	*Vertex_handle;*	A type for a handle to a vertex.
typedef typename DS::Face_handle	*Face_handle;*	A type for a handle to a face.
typedef typename DS::Vertex_circulator	*Vertex_circulator;*	A type for a circulator over vertices incident to a given vertex.
typedef typename DS::Face_circulator	*Face_circulator;*	A type for a circulator over faces incident to a given vertex.
typedef typename DS::Edge_circulator	*Edge_circulator;*	A type for a circulator over edges incident to a given vertex.
typedef typename DS::Vertex_iterator	*All_vertices_iterator;*	A type for an iterator over all vertices.
typedef typename DS::Face_iterator	*All_faces_iterator;*	A type for an iterator over all faces.
typedef typename DS::Edge_iterator	*All_edges_iterator;*	A type for an iterator over all edges.
Segment_Delaunay_graph_2<Gt,DS>:: Finite_vertices_iterator		A type for an iterator over finite vertices.
Segment_Delaunay_graph_2<Gt,DS>:: Finite_faces_iterator		A type for an iterator over finite faces.
Segment_Delaunay_graph_2<Gt,DS>:: Finite_edges_iterator		A type for an iterator over finite edges.

In addition to iterators and circulators for vertices and faces, iterators for sites are provided. In particular there are iterators for the set of input sites and the set of output sites. The set of input sites is the set of sites inserted by the user using the *insert* methods of this class. If a site is inserted multiple times, every instance of this site will be reported. The set of output sites is the set of sites in the segment Delaunay graph. The value type of these iterators is *Site_2*.

Segment_Delaunay_graph_2<Gt,DS>:: Input_sites_iterator

A type for a bidirectional iterator over all input sites.

Segment_Delaunay_graph_2<Gt,DS>:: Output_sites_iterator

A type for a bidirectional iterator over all output sites (the sites in the Delaunay graph).

Creation

In addition to the default and copy constructors the following constructors are defined:

Segment_Delaunay_graph_2<Gt,DS> sdg(Gt gt=Gt());

Creates the segment Delaunay graph using *gt* as geometric traits.

template< class Input_iterator >
Segment_Delaunay_graph_2<Gt,DS> sdg(Input_iterator first, Input_iterator beyond, Gt gt=Gt());

Creates the segment Delaunay graph using *gt* as geometric traits and inserts all sites in the range [*first*, *beyond*).
Precondition: *Input_iterator* must be a model of *InputIterator*. The value type of *Input_iterator* must be either *Point_2* or *Site_2*.

Access Functions

Geom_traits *sdg.geom_traits()*

Returns a reference to the segment Delaunay graph traits object.

int	*sdg.dimension()*	Returns the dimension of the segment Delaunay graph. The dimension is -1 if the graph contains no sites, 0 if the graph contains one site, 1 if it contains two sites and 2 if it contains three or more sites.
size_type	*sdg.number_of_vertices()*	Returns the number of finite vertices of the segment Delaunay graph.
size_type	*sdg.number_of_faces()*	Returns the number of faces (both finite and infinite) of the segment Delaunay graph.
size_type	*sdg.number_of_input_sites()*	Return the number of input sites.
size_type	*sdg.number_of_output_sites()*	Return the number of output sites. This is equal to the number of vertices in the segment Delaunay graph.
Face_handle	*sdg.infinite_face()*	Returns a face incident to the *infinite_vertex*.
Vertex_handle	*sdg.infinite_vertex()*	Returns the *infinite_vertex*.
Vertex_handle	*sdg.finite_vertex()*	Returns a vertex distinct from the *infinite_vertex*. *Precondition*: The number of sites in the segment Delaunay graph must be at least one.
Data_structure	*sdg.data_structure()*	Returns a reference to the segment Delaunay graph data structure object.
Data_structure	*sdg.tds()*	Same as *data_structure()*. It has been added for compliance to the *DelaunayGraph_2* concept.
Point_container	*sdg.point_container()*	Returns a reference to the point container object.

Traversal of the segment Delaunay graph

A segment Delaunay graph can be seen as a container of faces and vertices. Therefore the *Segment_Delaunay_graph_2<Gt,DS>* class provides several iterators and circulators that allow to traverse it (completely or partially).

Face, Edge and Vertex Iterators

The following iterators allow respectively to visit finite faces, finite edges and finite vertices of the segment Delaunay graph. These iterators are non-mutable, bidirectional and their value types are respectively *Face*, *Edge* and *Vertex*. They are all invalidated by any change in the segment Delaunay graph.

Finite_vertices_iterator	*sdg.finite_vertices_begin()*	Starts at an arbitrary finite vertex.
Finite_vertices_iterator	*sdg.finite_vertices_end()*	Past-the-end iterator.
Finite_edges_iterator	*sdg.finite_edges_begin()*	Starts at an arbitrary finite edge.
Finite_edges_iterator	*sdg.finite_edges_end()*	Past-the-end iterator.
Finite_faces_iterator	*sdg.finite_faces_begin()*	Starts at an arbitrary finite face.
Finite_faces_iterator	*sdg.finite_faces_end() const*	Past-the-end iterator.

The following iterators allow respectively to visit all (both finite and infinite) faces, edges and vertices of the segment Delaunay graph. These iterators are non-mutable, bidirectional and their value types are respectively *Face*, *Edge* and *Vertex*. They are all invalidated by any change in the segment Delaunay graph.

All_vertices_iterator	*sdg.all_vertices_begin()*	Starts at an arbitrary vertex.
All_vertices_iterator	*sdg.all_vertices_end()*	Past-the-end iterator.
All_edges_iterator	*sdg.all_edges_begin()*	Starts at an arbitrary edge.

All_edges_iterator	*sdg.all_edges_end()*	Past-the-end iterator.

All_faces_iterator	*sdg.all_faces_begin()*	Starts at an arbitrary face.
All_faces_iterator	*sdg.all_faces_end()*	Past-the-end iterator.

Site iterators

The following iterators allow respectively to visit all sites. These iterators are non-mutable, bidirectional and their value type is *Site_2*. They are all invalidated by any change in the segment Delaunay graph.

Input_sites_iterator	*sdg.input_sites_begin()*	Starts at an arbitrary input site.
Input_sites_iterator	*sdg.input_sites_end()*	Past-the-end iterator.
Output_sites_iterator	*sdg.output_sites_begin()*	Starts at an arbitrary output site.
Output_sites_iterator	*sdg.output_sites_end()*	Past-the-end iterator.

Face, Edge and Vertex Circulators

The *Segment_Delaunay_graph_2<Gt,DS>* class also provides circulators that allow to visit respectively all faces or edges incident to a given vertex or all vertices adjacent to a given vertex. These circulators are non-mutable and bidirectional. The operator *operator++* moves the circulator counterclockwise around the vertex while the *operator--* moves clockwise. A face circulator is invalidated by any modification of the face pointed to. An edge circulator is invalidated by any modification in one of the two faces incident to the edge pointed to. A vertex circulator is invalidated by any modification in any of the faces adjacent to the vertex pointed to.

Face_circulator *sdg.incident_faces(Vertex_handle v)*

Starts at an arbitrary face incident to v.

Face_circulator *sdg.incident_faces(Vertex_handle v, Face_handle f)*

Starts at face f.
Precondition: Face f is incident to vertex v.

Edge_circulator *sdg.incident_edges(Vertex_handle v)*

Starts at an arbitrary edge incident to v.

Edge_circulator *sdg.incident_edges(Vertex_handle v, Face_handle f)*

Starts at the first edge of f incident to v, in counterclockwise order around v.
Precondition: Face f is incident to vertex v.

Vertex_circulator *sdg.incident_vertices(Vertex_handle v)*

Starts at an arbitrary vertex incident to v.

Vertex_circulator *sdg.incident_vertices(Vertex_handle v, Face_handle f)*

Starts at the first vertex of f adjacent to v in counterclockwise order around v.
Precondition: Face f is incident to vertex v.

Traversal of the Convex Hull

Applied on the *infinite_vertex* the above methods allow to visit the vertices on the convex hull and the infinite edges and faces. Note that a counterclockwise traversal of the vertices adjacent to the *infinite_vertex* is a clockwise traversal of the convex hull.

Vertex_circulator *sdg.incident_vertices(sdg.infinite_vertex())*
Vertex_circulator *sdg.incident_vertices(sdg.infinite_vertex(), Face_handle f)*
Face_circulator *sdg.incident_faces(sdg.infinite_vertex())*
Face_circulator *sdg.incident_faces(sdg.infinite_vertex(), Face_handle f)*
Edge_circulator *sdg.incident_edges(sdg.infinite_vertex())*
Edge_circulator *sdg.incident_edges(sdg.infinite_vertex(), Face_handle f)*

Predicates

The class *Segment_Delaunay_graph_2<Gt,DS>* provides methods to test the finite or infinite character of any feature.

bool *sdg.is_infinite(Vertex_handle v) const*

true, iff *v* is the *infinite_vertex*.

bool *sdg.is_infinite(Face_handle f) const* *true*, iff face *f* is infinite.

bool *sdg.is_infinite(Face_handle f, int i) const*

true, iff edge *(f,i)* is infinite.

bool *sdg.is_infinite(Edge e) const* *true*, iff edge *e* is infinite.

bool *sdg.is_infinite(Edge_circulator ec) const*

true, iff edge **ec* is infinite.

Insertion

template< class Input_iterator >
size_type *sdg.insert(Input_iterator first, Input_iterator beyond)*

Inserts the sites in the range [*first*,*beyond*). The number of additional sites inserted in the Delaunay graph is returned. *Input_iterator* must be a model of *InputIterator* and its value type must be either *Point_2* or *Site_2*.

template< class Input_iterator >
size_type *sdg.insert(Input_iterator first, Input_iterator beyond, Tag_false)*

Same as the previous method. *Input_iterator* must be a model of *InputIterator* and its value type must be either *Point_2* or *Site_2*.

template< class Input_iterator >

size_type *sdg.insert(Input_iterator first, Input_iterator beyond, Tag_true)*

Inserts the sites in the range [*first*,*beyond*) after performing a random shuffle on them. The number of additional sites inserted in the Delaunay graph is returned. *Input_iterator* must be a model of *InputIterator* and its value type must be either *Point_2* or *Site_2*.

Vertex_handle *sdg.insert(Point_2 p)* Inserts the point *p* in the segment Delaunay graph. If *p* has already been inserted, then the vertex handle of its already inserted copy is returned. If *p* has not been inserted yet, the vertex handle of *p* is returned.

Vertex_handle *sdg.insert(Point_2 p, Vertex_handle vnear)*

Inserts *p* in the segment Delaunay graph using the site associated with *vnear* as an estimate for the nearest neighbor of *p*. The vertex handle returned has the same semantics as the vertex handle returned by the method *Vertex_handle insert(Point_2 p)*.

Vertex_handle *sdg.insert(Point_2 p1, Point_2 p2)*

Inserts the closed segment with endpoints *p1* and *p2* in the segment Delaunay graph. If the segment has already been inserted in the Delaunay graph then the vertex handle of its already inserted copy is returned. If the segment does not intersect any segment in the existing diagram, the vertex handle corresponding to its corresponding open segment is returned. Finally, if the segment intersects other segments in the existing Delaunay graph, the vertex handle to one of its open subsegments is returned.

Vertex_handle *sdg.insert(Point_2 p1, Point_2 p2, Vertex_handle vnear)*

Inserts the segment whose endpoints are *p1* and *p2* in the segment Delaunay graph using the site associated with *vnear* as an estimate for the nearest neighbor of *p1*. The vertex handle returned has the same semantics as the vertex handle returned by the method *Vertex_handle insert(Point_2 p1, Point_2 p2)*.

Vertex_handle *sdg.insert(Site_2 s)* Inserts the site *s* in the segment Delaunay graph. The vertex handle returned has the same semantics as the vertex handle returned by the methods *Vertex_handle insert(Point_2 p)* and *Vertex_handle insert(Point_2 p1, Point_2 p2)*, depending on whether *s* represents a point or a segment respectively.
Precondition: *s.is_input()* must be *true*.

Vertex_handle *sdg.insert(Site_2 s, Vertex_handle vnear)*

Inserts *s* in the segment Delaunay graph using the site associated with *vnear* as an estimate for the nearest neighbor of *s*, if *s* is a point, or the first endpoint of *s*, if *s* is a segment. The vertex handle returned has the same semantics as the vertex handle returned by the method *Vertex_handle insert(Site_2 s)*.
Precondition: *s.is_input()* must be *true*.

Nearest neighbor location

Vertex_handle *sdg.nearest_neighbor(Point_2 p)*

Finds the nearest neighbor of the point *p*. In other words it finds the site whose segment Voronoi diagram cell contains *p*. Ties are broken arbitrarily and one of the nearest neighbors of *p* is returned. If there are no sites in the segment Delaunay graph *Vertex_handle()* is returned.

Vertex_handle *sdg.nearest_neighbor(Point_2 p, Vertex_handle vnear)*

Finds the nearest neighbor of the point *p* using the site associated with *vnear* as an estimate for the nearest neighbor of *p*. Ties are broken arbitrarily and one of the nearest neighbors of *p* is returned. If there are no sites in the segment Delaunay graph *Vertex_handle()* is returned.

I/O

template < class Stream >
Stream& *sdg.draw_dual(Stream& str)*

Draws the segment Voronoi diagram to the stream *str*. The following operators must be defined:
Stream& operator<<(Stream&, Gt::Segment_2)
Stream& operator<<(Stream&, Gt::Ray_2)
Stream& operator<<(Stream&, Gt::Line_2)

template < class Stream >
Stream& *sdg.draw_skeleton(Stream& str)*

Draws the segment Voronoi diagram to the stream *str*, except the edges of the diagram corresponding to a segment and its endpoints. The following operators must be defined:
Stream& operator<<(Stream&, Gt::Segment_2)
Stream& operator<<(Stream&, Gt::Ray_2)
Stream& operator<<(Stream&, Gt::Line_2)

template< class Stream >
Stream& *sdg.draw_dual_edge(Edge e, Stream& str)*

Draws the edge *e* of the segment Voronoi diagram to the stream *str*. The following operators must be defined:
Stream& operator<<(Stream&, Gt::Segment_2)
Stream& operator<<(Stream&, Gt::Ray_2)
Stream& operator<<(Stream&, Gt::Line_2)
Precondition: *e* must be a finite edge.

template< class Stream >
Stream& *sdg.draw_dual_edge(Edge_circulator ec, Stream& str)*

Draws the edge **ec* of the segment Voronoi diagram to the stream *str*. The following operators must be defined:
Stream& operator<<(Stream&, Gt::Segment_2)
Stream& operator<<(Stream&, Gt::Ray_2)
Stream& operator<<(Stream&, Gt::Line_2)
Precondition: **ec* must be a finite edge.

template< class Stream >

Stream& *sdg.draw_dual_edge(All_edges_iterator eit, Stream& str)*

Draws the edge **eit* of the segment Voronoi diagram to the stream *str*. The following operators must be defined:
Stream& operator<<(Stream&, Gt::Segment_2)
Stream& operator<<(Stream&, Gt::Ray_2)
Stream& operator<<(Stream&, Gt::Line_2)
Precondition: **eit* must be a finite edge.

template< class Stream >
Stream& *sdg.draw_dual_edge(Finite_edges_iterator eit, Stream& str)*

Draws the edge **eit* of the segment Voronoi diagram to the stream *str*. The following operators must be defined:
Stream& operator<<(Stream&, Gt::Segment_2)
Stream& operator<<(Stream&, Gt::Ray_2)
Stream& operator<<(Stream&, Gt::Line_2)

void	*sdg.file_output(std::ostream& os)*	Writes the current state of the segment Delaunay graph to an output stream. In particular, all sites in the diagram are written to the stream (represented through appropriate input sites), as well as the underlying combinatorial data structure.
void	*sdg.file_input(std::istream& is)*	Reads the state of the segment Delaunay graph from an input stream.
std::ostream&	*std::ostream& os << sdg*	Writes the current state of the segment Delaunay graph to an output stream.
std::istream&	*std::istream& is >> sdg*	Reads the state of the segment Delaunay graph from an input stream.

Validity check

bool *sdg.is_valid(bool verbose = false, int level = 1)*

Checks the validity of the segment Delaunay graph. If *verbose* is *true* a short message is sent to *std::cerr*. If *level* is 0, only the data structure is validated. If *level* is 1, then both the data structure and the segment Delaunay graph are validated. Negative values of *level* always return true, and values greater than 1 are equivalent to *level* being 1.

Miscellaneous

void	*sdg.clear()*	Clears all contents of the segment Delaunay graph.
void	*sdg.swap(other)*	The segment Delaunay graphs *other* and *sdg* are swapped. *sdg.swap(other)* should be preferred to *sdg= other* or to *sdg(other)* if *other* is deleted afterwards.

See Also

DelaunayGraph_2
SegmentDelaunayGraphTraits_2

SegmentDelaunayGraphDataStructure_2
SegmentDelaunayGraphVertexBase_2
TriangulationFaceBase_2
CGAL::Segment_Delaunay_graph_hierarchy_2<Gt,STag,DS>
CGAL::Segment_Delaunay_graph_traits_2<K,MTag>
CGAL::Segment_Delaunay_graph_traits_without_intersections_2<K,MTag>
CGAL::Segment_Delaunay_graph_filtered_traits_2<CK,CM,EK,EM,FK,FM>
CGAL::Segment_Delaunay_graph_filtered_traits_without_intersections_2<CK,CM,EK,EM,FK,FM>
CGAL::Triangulation_data_structure_2<Vb,Fb>
CGAL::Segment_Delaunay_graph_vertex_base_2<Gt,SSTag>
CGAL::Triangulation_face_base_2<Gt>

SegmentDelaunayGraphSite_2

Definition

The concept *SegmentDelaunayGraphSite_2* provides the requirements for the sites of a segment Delaunay graph.

Refines

DefaultConstructible
CopyConstructible
Assignable

Types

SegmentDelaunayGraphSite_2:: Point_2	The point type.
SegmentDelaunayGraphSite_2:: Segment_2	The segment type.
SegmentDelaunayGraphSite_2:: FT	The field number type.
SegmentDelaunayGraphSite_2:: RT	The ring number type.

Creation

In addition to the default and copy constructors the following static methods are available for constructing sites:

SegmentDelaunayGraphSite_2 *s.construct_site_2(Point_2 p)*

Constructs a site from a point: the site represents the point *p*.

SegmentDelaunayGraphSite_2 *s.construct_site_2(Point_2 p1, Point_2 p2)*

Constructs a site from two points: the site represents the (open) segment *(p1,p2)*.

SegmentDelaunayGraphSite_2 *s.construct_site_2(Point_2 p1, Point_2 p2, Point_2 q1, Point_2 q2)*

Constructs a site from four points: the site represents the point of intersection of the segments *(p1,p2)* and *(q1,q2)*.

SegmentDelaunayGraphSite_2 *s.construct_site_2(Point_2 p1, Point_2 p2, Point_2 q1, Point_2 q2, bool b)*

Constructs a site from four points and a boolean: the site represents a segment. If *b* is *true* the endpoints are *p1* and $p_\times$, otherwise $p_\times$ and *p2*. $p_\times$ is the point of intersection of the segments *(p1,p2)*,*(q1,q2)*.

SegmentDelaunayGraphSite_2 *s.construct_site_2(Point_2 p1,*
Point_2 p2,
Point_2 q1,
Point_2 q2,
Point_2 r1,

Point_2 r2)

Constructs a site from six points: the site represents the segment with endpoints the points of intersection of the pairs of segments *(p1,p2)*,*(q1,q2)* and *(p1,p2)*,*(r1,r2)*.

Predicates

bool	*s.is_defined()*	Returns *true* if the site represents a valid point or segment.
bool	*s.is_point()*	Returns *true* if the site represents a point.
bool	*s.is_segment()*	Returns *true* if the site represents a segment.
bool	*s.is_input()*	Returns *true* if the site represents an input point or a segment defined by two input points. Returns *false* if it represents a point of intersection of two segments, or if it represents a segment, at least one endpoint of which is a point of intersection of two segments.
bool	*s.is_input(unsigned int i)*	Returns *true* if the *i*-th endpoint of the site is an input point. Returns *false* if the *i*-th endpoint of the site is the intersection of two segments. *Precondition*: *i* must be at most 1, and *s.is_segment()* must be *true*.

Access Functions

Point_2 *s.point() const*

Returns the point represented by the site *s*.
Precondition: *s.is_point()* must be *true*.

Segment_2 *s.segment() const*

Returns the segment represented by the site *s*.
Precondition: *s.is_segment()* must be *true*.

Point_2 *s.source() const*

Returns the source endpoint of the segment. Note that this method can construct an inexact point if the number type used is inexact.
Precondition: *s.is_segment()* must be *true*.

Point_2 *s.target() const*

Returns the target endpoint of the segment. Note that this method can construct an inexact point if the number type used is inexact.
Precondition: *s.is_segment()* must be *true*.

SegmentDelaunayGraphSite_2 *s.supporting_site()*

Returns a segment site object representing the segment that supports the segment represented by the site. Both endpoints of the returned site are input points.
Precondition: *s.is_segment()* must be *true*.

SegmentDelaunayGraphSite_2 *s.supporting_site(unsigned int i)*

Returns a segment site object representing the *i*-th segment that supports the point of intersection represented by the site. Both endpoints of the returned site are input points.
Precondition: *i* must be at most 1, *s.is_point()* must be *true* and *s.is_input()* must be *false*.

SegmentDelaunayGraphSite_2 *s.crossing_site(unsigned int i)*

Returns a segment site object representing the *i*-th segment that supports the *i*-th endpoint of the site which is not the supporting segment of the site. Both endpoints of the returned site are input points.
Precondition: *i* must be at most 1, *s.is_segment()* must be *true* and *s.is_input(i)* must be *false*.

SegmentDelaunayGraphSite_2 *s.source_site()*

Returns a point site object representing the source point of the site.
Precondition: *s.is_segment()* must be *true*.

SegmentDelaunayGraphSite_2 *s.target_site()*

Returns a point site object representing the target point of the site.
Precondition: *s.is_segment()* must be *true*.

Point_2 *s.source_of_supporting_site()*

Returns the source point of the supporting site of the this site.
Precondition: *is_segment()* must be *true*.

Point_2 *s.target_of_supporting_site()*

Returns the target point of the supporting site of the this site.
Precondition: *is_segment()* must be *true*.

Point_2 *s.source_of_supporting_site(unsigned int i)*

Returns the source point of the *i*-th supporting site of the this site.
Precondition: *is_point()* must be *true*, *is_input()* must be *false* and *i* must either be *0* or *1*.

Point_2 *s.target_of_supporting_site(unsigned int i)*

Returns the target point of the *i*-th supporting site of the this site.
Precondition: *is_point()* must be *true*, *is_input()* must be *false* and *i* must either be *0* or *1*.

Point_2 *s.source_of_crossing_site(unsigned int i)*

Returns the source point of the *i*-th crossing site of the this site.
Precondition: *is_segment()* must be *true*, *is_input(i)* must be *false* and *i* must either be *0* or *1*.

Point_2 *s.target_of_crossing_site(unsigned int i)*

Returns the target point of the *i*-th supporting site of the this site.
Precondition: *is_segment()* must be *true*, *is_input(i)* must be *false* and *i* must either be *0* or *1*.

Has Models

CGAL::Segment_Delaunay_graph_site_2<K>

See Also

SegmentDelaunayGraphTraits_2
CGAL::Segment_Delaunay_graph_site_2<K>

CGAL::Segment_Delaunay_graph_traits_2<K,MTag>
CGAL::Segment_Delaunay_graph_traits_without_intersections_2<K,MTag>
CGAL::Segment_Delaunay_graph_filtered_traits_2<CK,CM,EK,EM,FK,FM>
CGAL::Segment_Delaunay_graph_filtered_traits_without_intersections_2<CK,CM,EK,EM,FK,FM>

Class

CGAL::Segment_Delaunay_graph_site_2<K>

Definition

The class *Segment_Delaunay_graph_site_2<K>* is a model for the concept *SegmentDelaunayGraphSite_2*. It is parametrized by a template parameter *K* which must be a model of the *Kernel* concept.

#include <CGAL/Segment_Delaunay_graph_site_2.h>

Is Model for the Concepts

SegmentDelaunayGraphSite_2

Types

The class *Segment_Delaunay_graph_site_2<K>* introduces the following type in addition to the types in the concept *SegmentDelaunayGraphSite_2*.

typedef K	*Kernel;*	A type for the template parameter *K*.

See Also

Kernel
SegmentDelaunayGraphSite_2
CGAL::Segment_Delaunay_graph_traits_2<K,MTag>
CGAL::Segment_Delaunay_graph_traits_without_intersections_2<K,MTag>
CGAL::Segment_Delaunay_graph_filtered_traits_2<CK,CM,EK,EM,FK,FM>
CGAL::Segment_Delaunay_graph_filtered_traits_without_intersections_2<CK,CM,EK,EM,FK,FM>

SegmentDelaunayGraphStorageSite_2

Concept

Definition

The concept *SegmentDelaunayGraphStorageSite_2* provides the requirements for the storage sites of a segment Delaunay graph. The storage sites are sites that are used to store the information of a site in a more compact form (that uses less storage). This is achieved by storing handles to points instead of points.

Refines

DefaultConstructible
CopyConstructible
Assignable

Types

SegmentDelaunayGraphStorageSite_2:: Site_2		The site type.
typedef typename std::set<typename Site_2::Point_2>::iterator	*Point_handle;*	The type for a handle to a point.

Creation

In addition to the default and copy constructors, the following static methods should be available for constructing sites:

SegmentDelaunayGraphStorageSite_2 *ss.construct_storage_site_2(Point_handle hp)*

Constructs a storage site from a point handle. The storage site represents the point associated with the point handle *hp*.

SegmentDelaunayGraphStorageSite_2 *ss.construct_storage_site_2(Point_handle hp1, Point_handle hp2)*

Constructs a storage site from two point handles. The storage site represents the segment the endpoints of which are the points associated with the point handles *hp1* and *hp2*.

SegmentDelaunayGraphStorageSite_2 *ss.construct_storage_site_2(Point_handle hp1, Point_handle hp2, Point_handle hq1, Point_handle hq2)*

Constructs a storage site from four point handles. The storage site represents the point of intersection of the segments the endpoints of which are the points associated with the point handles *hp1*, *hp2* and *hq1* and *hq2*, respectively.

SegmentDelaunayGraphStorageSite_2 *ss.construct_storage_site_2(Point_handle hp1, Point_handle hp2, Point_handle hq1, Point_handle hq2, bool b)*

Constructs a site from four point handles and a boolean. The storage site represents a segment. If *b* is *true*, the first endpoint of the segment is the point associated with the handle *hp1* and the second endpoint is the point of intersection of the segments the endpoints of which are the point associated with the point handles *hp1*, *hp2* and *hq1*, *hq2*, respectively. If *b* is *false*, the first endpoint of the represented segment is the one mentioned above, whereas the second endpoint if the point associated with the point handle *hp2*.

SegmentDelaunayGraphStorageSite_2 *ss.construct_storage_site_2(Point_handle hp1, Point_handle hp2, Point_handle hq1, Point_handle hq2, Point_handle hr1, Point_handle hr2)*

Constructs a storage site from six point handles. The storage site represents of segment the endpoints of which are points of intersection of two pairs of segments, the endpoints of which are *hp1*, *hp2*/*hq1*, *hq2* and *hp1*, *hp2*/*hr1*, *hr2*, respectively.

Predicates

bool	*ss.is_defined()*	Returns *true* if the storage site represents a valid point or segment.
bool	*ss.is_point()*	Returns *true* if the storage site represents a point.
bool	*ss.is_segment()*	Returns *true* if the storage site represents a segment.
bool	*ss.is_input()*	Returns *true* if the storage site represents an input point or a segment defined by two input points. Returns *false* if it represents a point of intersection of two segments, or if it represents a segment, at least one endpoint of which is a point of intersection of two segments.
bool	*ss.is_input(unsigned int i)*	Returns *true* if the *i*-th endpoint of the corresponding site is an input point. Returns *false* if the *i*-th endpoint of the corresponding site is the intersection of two segments. *Precondition*: *i* must be at most 1, and *ss.is_segment()* must be *true*.

Access Functions

SegmentDelaunayGraphStorageSite_2 *ss.supporting_site()*

Returns a storage site object representing the segment that supports the segment represented by the storage site. The returned storage site represents a site, both endpoints of which are input points.
Precondition: *ss.is_segment()* must be *true*.

SegmentDelaunayGraphStorageSite_2 *ss.source_site()*

Returns a storage site that represents the first endpoint of the represented segment.
Precondition: *ss.is_segment()* must be *true*.

SegmentDelaunayGraphStorageSite_2 *ss.target_site()*

Returns a storage site that represents the second endpoint of the represented segment.
Precondition: *ss.is_segment()* must be *true*.

SegmentDelaunayGraphStorageSite_2 *ss.supporting_site(unsigned int i)*

Returns a storage site object representing the i-th segment that supports the point of intersection represented by the storage site. The returned storage site represents a site, both endpoints of which are input points.
Precondition: i must be at most 1, *ss.is_point()* must be *true* and *ss.is_input()* must be *false*.

SegmentDelaunayGraphStorageSite_2 *ss.crossing_site(unsigned int i)*

Returns a storage site object representing the i-th segment that supports the i-th endpoint of the site which is not the supporting segment of the site. The returned storage site represents a site, both endpoints of which are input points.
Precondition: i must be at most 1, *ss.is_segment()* must be *true* and *ss.is_input(i)* must be *false*.

Site_2 *ss.site()*

Returns the site represented by the storage site.

Point_handle *ss.point()*

Returns a handle associated with the represented point.
Precondition: *is_point()* and *is_input()* must both be *true*.

Point_handle *ss.source_of_supporting_site()*

Returns a handle to the source point of the supporting site of the this site.
Precondition: *is_segment()* must be *true*.

Point_handle *ss.target_of_supporting_site()*

Returns a handle to the target point of the supporting site of the this site.
Precondition: *is_segment()* must be *true*.

Point_handle *ss.source_of_supporting_site(unsigned int i)*

Returns a handle to the source point of the i-th supporting site of the this site.
Precondition: *is_point()* must be *true*, *is_input()* must be *false* and i must either be *0* or *1*.

Point_handle *ss.target_of_supporting_site(unsigned int i)*

Returns a handle to the target point of the i-th supporting site of the this site.
Precondition: *is_point()* must be *true*, *is_input()* must be *false* and i must either be *0* or *1*.

Point_handle *ss.source_of_crossing_site(unsigned int i)*

Returns a handle to the source point of the *i*-th crossing site of the this site.
Precondition: *is_segment()* must be *true*, *is_input(i)* must be *false* and *i* must either be *0* or *1*.

Point_handle *ss.target_of_crossing_site(unsigned int i)*

Returns a handle to the target point of the *i*-th supporting site of the this site.
Precondition: *is_segment()* must be *true*, *is_input(i)* must be *false* and *i* must either be *0* or *1*.

Has Models

CGAL::Segment_Delaunay_graph_storage_site_2<Gt>

See Also

SegmentDelaunayGraphTraits_2
CGAL::Segment_Delaunay_graph_site_2<K>
CGAL::Segment_Delaunay_graph_storage_site_2<Gt>
CGAL::Segment_Delaunay_graph_traits_2<K,MTag>
CGAL::Segment_Delaunay_graph_traits_without_intersections_2<K,MTag>
CGAL::Segment_Delaunay_graph_filtered_traits_2<CK,CM,EK,EM,FK,FM>
CGAL::Segment_Delaunay_graph_filtered_traits_without_intersections_2<CK,CM,EK,EM,FK,FM>

CGAL::Segment_Delaunay_graph_storage_site_2<Gt>

Class

Definition

The class *Segment_Delaunay_graph_storage_site_2<Gt>* is a model for the concept *SegmentDelaunayGraphStorageSite_2*. It is parametrized by a single template parameter *Gt*, which must be a model of the *SegmentDelaunayGraphTraits_2* concept.

#include <CGAL/Segment_Delaunay_graph_storage_site_2.h>

Is Model for the Concepts

SegmentDelaunayGraphStorageSite_2

Types

The class *Segment_Delaunay_graph_storage_site_2<Gt>* introduces the following type in addition to the types in the concept *SegmentDelaunayGraphStorageSite_2*.

typedef Gt	*Geom_traits;*	A type for the template parameter *Gt*.

See Also

SegmentDelaunayGraphSite_2
SegmentDelaunayGraphTraits_2
CGAL::Segment_Delaunay_graph_site_2<K>
CGAL::Segment_Delaunay_graph_traits_2<K,MTag>
CGAL::Segment_Delaunay_graph_traits_without_intersections_2<K,MTag>
CGAL::Segment_Delaunay_graph_filtered_traits_2<CK,CM,EK,EM,FK,FM>
CGAL::Segment_Delaunay_graph_filtered_traits_without_intersections_2<CK,CM,EK,EM,FK,FM>

SegmentDelaunayGraphDataStructure_2

Definition

The concept *SegmentDelaunayGraphDataStructure_2* refines the concept *ApolloniusGraphDataStructure_2*. In addition it provides two methods for the merging of two vertices joined by an edge of the data structure, and the splitting of a vertex into two. The method that merges two vertices, called *join_vertices* identifies the two vertices and deletes their common two faces. The method that splits a vertex, called *split_vertex* introduces a new vertex that shares an edge and two faces with the old vertex (see figure below). Notice that the *join_vertices* and *split_vertex* operations are complementary, in the sense that one reverses the action of the other.

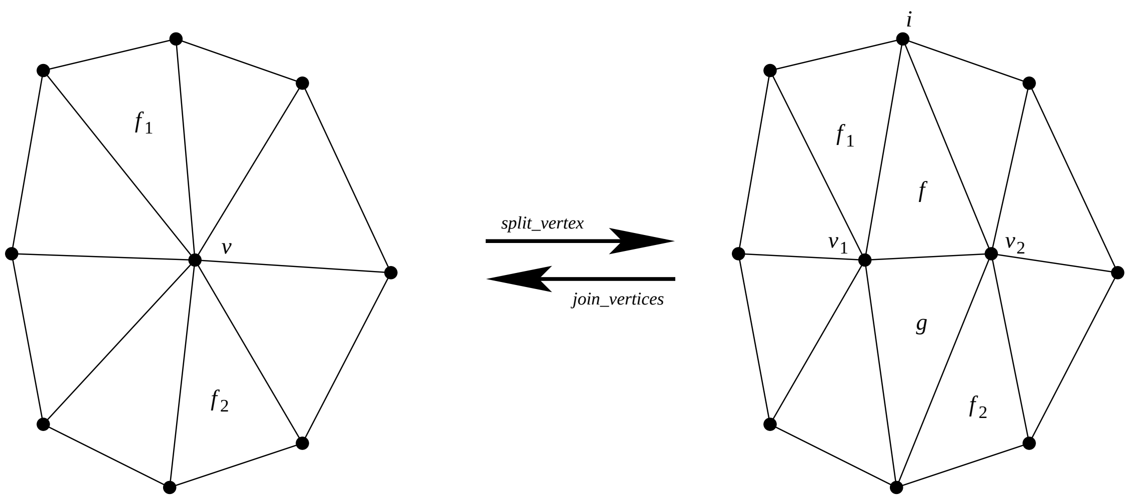


Figure 44.3: The join and split operations. Left to right: The vertex v is split into v_1 and v_2. The faces f and g are inserted after f_1 and f_2, respectively, in the counter-clockwise sense. The vertices v_1, v_2 and the faces f and g are returned as a *boost tuple* in that order. Right to left: The edge *(f,i)* is collapsed, and thus the vertices v_1 and v_2 are joined. The vertex v is returned.

We only describe the additional requirements with respect to the *ApolloniusGraphDataStructure_2* concept.

Refines

ApolloniusGraphDataStructure_2

Modification

Vertex_handle	*sdgds.join_vertices(Face_handle f, int i)*	Joins the vertices that are endpoints of the edge *(f,i)*. It returns a vertex handle to common vertex.

boost::tuples::tuple<Vertex_handle, Vertex_handle, Face_handle, Face_handle>

sdgds.split_vertex(Vertex_handle v, Face_handle f1, Face_handle f2)

Splits the vertex *v* into two vertices *v1* and *v2*. The common faces *f* and *g* of *v1* and *v2* are created after (in the counter-clockwise sense) the faces *f1* and *f2*. The 4-tuple *(v1,v2,f,g)* is returned (see Fig. 44.6).

Has Models

CGAL::Triangulation_data_structure_2<Vb,Fb>

See Also

TriangulationDataStructure_2
ApolloniusGraphDataStructure_2
SegmentDelaunayGraphVertexBase_2
TriangulationFaceBase_2

Concept

SegmentDelaunayGraphVertexBase_2

Definition

The concept *SegmentDelaunayGraphVertexBase_2* describes the requirements for the vertex base class of the *SegmentDelaunayGraphDataStructure_2* concept. A vertex stores a site of the segment Delaunay graph and provides access to one of its incident faces through a *Face_handle*.

Refines

DefaultConstructible
CopyConstructible
Assignable

Types

SegmentDelaunayGraphVertexBase_2:: Geom_traits	A type for the geometric traits that defines the site. *Precondition*: The type *Geom_traits* must define the type *Site_2*.
SegmentDelaunayGraphVertexBase_2:: Site_2	A type for the site. This type must coincide with the type *Geom_traits::Site_2*.
SegmentDelaunayGraphVertexBase_2:: Storage_site_tag	A type that indicates what kind of storage type to use. *Storage_site_tag* must either be *CGAL::Tag_true* or *CGAL::Tag_false*.
SegmentDelaunayGraphVertexBase_2:: Storage_site_2	A type for the internal representation of sites. This type must satisfy the requirements of the concept *SegmentDelaunayGraphStorageSite_2*.
SegmentDelaunayGraphVertexBase_2:: Data_structure	A type for the underlying data structure, to which the vertex belongs to.
SegmentDelaunayGraphVertexBase_2:: Vertex_handle	A type for the vertex handle of the segment Delaunay graph data structure.
SegmentDelaunayGraphVertexBase_2:: Face_handle	A type for the face handle of the segment Delaunay graph data structure.

Creation

In addition to the default and copy constructors and following constructors are required:

SegmentDelaunayGraphVertexBase_2 v(Storage_site_2 ss);

Constructs a vertex associated with the site represented by the storage site *ss*.

SegmentDelaunayGraphVertexBase_2 v(Storage_site_2 ss, Face_handle f);

Constructs a vertex associated with the site represented by the storage site *ss*, and pointing to the face associated with the face handle *f*.

Access Functions

Storage_site_2	*v.storage_site()*	Returns the storage site representing the site.
Site_2	*v.site()*	Returns the site.
Face_handle	*v.face()*	Returns a handle to an incident face.

Setting

void	*v.set_site(Storage_site_2 ss)*	Sets the storage site.
void	*v.set_face(Face_handle f)*	Sets the incident face.

Checking

bool *v.is_valid(bool verbose, int level) const*

Performs any required tests on a vertex.

Has Models

CGAL::Segment_Delaunay_graph_vertex_base_2<Gt>.

See Also

SegmentDelaunayGraphDataStructure_2
SegmentDelaunayGraphTraits_2
SegmentDelaunayGraphSite_2
SegmentDelaunayGraphStorageSite_2
CGAL::Segment_Delaunay_graph_vertex_base_2<Gt>
CGAL::Segment_Delaunay_graph_site_2<K>
CGAL::Segment_Delaunay_graph_storage_site_2<Gt,SSTag>
CGAL::Triangulation_data_structure_2<Vb,Fb>

Class

CGAL::Segment_Delaunay_graph_vertex_base_2<Gt,SSTag>

Definition

The class *Segment_Delaunay_graph_vertex_base_2<Gt,SSTag>* provides a model for the *SegmentDelaunayGraphVertexBase_2* concept which is the vertex base required by the *SegmentDelaunayGraphDataStructure_2* concept. The class *Segment_Delaunay_graph_vertex_base_2<Gt,SSTag>* has two template arguments, the first being the geometric traits of the segment Delaunay graph and should be a model of the concept *SegmentDelaunayGraphTraits_2*. The second template argument indicates whether or not to use the simple storage site that does not support intersecting segments, or the full storage site, that supports intersecting segments. The possible values are *CGAL::Tag_true* and *CGAL::Tag_false*. *CGAL::Tag_true* indicates that the full storage site is to be used, whereas *CGAL::Tag_false* indicates that the simple storage site is to be used.

#include <CGAL/Segment_Delaunay_graph_vertex_base_2.h>

Is Model for the Concepts

SegmentDelaunayGraphVertexBase_2

See Also

SegmentDelaunayGraphVertexBase_2
SegmentDelaunayGraphDataStructure_2
SegmentDelaunayGraphTraits_2
CGAL::Triangulation_data_structure_2<Vb,Fb>
CGAL::Segment_Delaunay_graph_traits_2<K,MTag>
CGAL::Segment_Delaunay_graph_traits_without_intersections_2<K,MTag>
CGAL::Segment_Delaunay_graph_filtered_traits_2<CK,CM,EK,EM,FK,FM>
CGAL::Segment_Delaunay_graph_filtered_traits_without_intersections_2<CK,CM,EK,EM,FK,FM>

SegmentDelaunayGraphTraits_2

Definition

The concept *SegmentDelaunayGraphTraits_2* provides the traits requirements for the *Segment_Delaunay_graph_2<Gt,DS>* and *Segment_Delaunay_graph_hierarchy_2<Gt,STag,DS>* classes. In particular, it provides a type *Site_2*, which must be a model of the concept *SegmentDelaunayGraphSite_2*. It also provides constructions for sites and several function object types for the predicates.

Refines

DefaultConstructible
CopyConstructible
Assignable

Types

SegmentDelaunayGraphTraits_2:: Intersections_tag	Indicates or not whether the intersecting segments are to be supported. The tag must either be *CGAL::Tag_true* or *CGAL::Tag_false*.
SegmentDelaunayGraphTraits_2:: Site_2	A type for a site of the segment Delaunay graph. Must be a model of the concept *SegmentDelaunayGraphSite_2*.
SegmentDelaunayGraphTraits_2:: Point_2	A type for a point.
SegmentDelaunayGraphTraits_2:: Line_2	A type for a line. Only required if the segment Delaunay graph is inserted in a stream.
SegmentDelaunayGraphTraits_2:: Ray_2	A type for a ray. Only required if the segment Delaunay graph is inserted in a stream.
SegmentDelaunayGraphTraits_2:: Segment_2	A type for a segment. Only required if if the segment Delaunay graph is inserted in a stream.
SegmentDelaunayGraphTraits_2:: FT	A type for the field number type of sites, points, etc..
SegmentDelaunayGraphTraits_2:: RT	A type for the ring number type of sites, points, etc.
SegmentDelaunayGraphTraits_2:: Arrangement_type	An enumeration type that indicates the type of the arrangement of two sites. The possible values are *DISJOINT*, *IDENTICAL*, *CROSSING*, *TOUCHING_1*, *TOUCHING_2*, *TOUCHING_11*, *TOUCHING_12*, *TOUCHING_21*, *TOUCHING_22*, *OVERLAPPING_11*, *OVERLAPPING_12*, *OVERLAPPING_21*, *OVERLAPPING_22*, *INTERIOR*, *INTERIOR_1*, *INTERIOR_2*, *TOUCHING_11_INTERIOR_1*, *TOUCHING_11_INTERIOR_2*, *TOUCHING_12_INTERIOR_1*, *TOUCHING_12_INTERIOR_2*, *TOUCHING_21_INTERIOR_1*, *TOUCHING_21_INTERIOR_2*, *TOUCHING_22_INTERIOR_1*, *TOUCHING_22_INTERIOR_2*. A detailed description of the meaning of these values is shown the end of the reference manual for this concept. **(to be done)**

SegmentDelaunayGraphTraits_2:: Object_2 — A type representing different types of objects in two dimensions, namely: *Point_2*, *Site_2*, *Line_2*, *Ray_2* and *Segment_2*.

SegmentDelaunayGraphTraits_2:: Assign_2 — Must provide *template <class T> bool operator() (T& t, Object_2 o)* which assigns *o* to *t* if *o* was constructed from an object of type *T*. Returns *true*, if the assignment was possible.

SegmentDelaunayGraphTraits_2:: Construct_object_2 — Must provide *template <class T> Object_2 operator()(T t)* that constructs an object of type *Object_2* that contains *t* and returns it.

SegmentDelaunayGraphTraits_2:: Construct_svd_vertex_2

A constructor for a point of the segment Voronoi diagram equidistant from three sites. Must provide *Point_2 operator()(Site_2 s1, Site_2 s2, Site_2 s3)*, which constructs a point equidistant from the sites *s1*, *s2* and *s3*.

SegmentDelaunayGraphTraits_2:: Compare_x_2

A predicate object type. Must provide *Comparison_result operator()(Site_2 s1, Site_2 s2)*, which compares the *x*-coordinates of the points represented by the sites *s1* and *s2*.
Precondition: *s1* and *s2* must be points.

SegmentDelaunayGraphTraits_2:: Compare_y_2

A predicate object type. Must provide *Comparison_result operator()(Site_2 s1, Site_2 s2)*, which compares the *y*-coordinates of the points represented by the sites *s1* and *s2*.
Precondition: *s1* and *s2* must be points.

SegmentDelaunayGraphTraits_2:: Orientation_2

A predicate object type. Must provide *Orientation operator()(Site_2 s1, Site_2 s2, Site_2 s3)*, which performs the usual orientation test for three points. *s1*, *s2* and *s3*.
Precondition: the sites *s1*, *s2* and *s3* must be points.

SegmentDelaunayGraphTraits_2:: Equal_2

A predicate object type. Must provide *bool operator()(Site_2 s1, Site_2 s2)*, which determines is the points represented by the sites *s1* and *s2* are identical.
Precondition: *s1* and *s2* must be points.

SegmentDelaunayGraphTraits_2:: Are_parallel_2

A predicate object type. Must provide *bool operator()(Site_2 s1, Site_2 s2)*, which determines is the segments represented by the sites *s1* and *s2* are parallel.
Precondition: *s1* and *s2* must be segments.

SegmentDelaunayGraphTraits_2:: Oriented_side_of_bisector_2

A predicate object type. Must provide *Oriented_side operator()(Site_2 s1, Site_2 s2, Point_2 p)*, which returns the oriented side of the bisector of *s1* and *s2* that contains *p*. Returns *ON_POSITIVE_SIDE* if *p* lies in the half-space of *s1* (i.e., *p* is closer to *s1* than *s2*); returns *ON_NEGATIVE_SIDE* if *p* lies in the half-space of *s2*; returns *ON_ORIENTED_BOUNDARY* if *p* lies on the bisector of *s1* and *s2*.

SegmentDelaunayGraphTraits_2:: Vertex_conflict_2

A predicate object type. Must provide *Sign operator()(Site_2 s1, Site_2 s2, Site_2 s3, Site_2 q)*, which returns the sign of the distance of q from the Voronoi circle of *s1*, *s2*, *s3* (the Voronoi circle of three sites *s1*, *s2*, *s3* is a circle co-tangent to all three sites, that touches them in that order as we walk on its circumference in the counter-clockwise sense).
Precondition: the Voronoi circle of *s1*, *s2*, *s3* must exist.
Must also provide *Sign operator()(Site_2 s1, Site_2 s2, Site_2 q)*, which returns the sign of the distance of q from the bitangent line of *s1*, *s2* (a degenerate Voronoi circle, with its center at infinity).

SegmentDelaunayGraphTraits_2:: Finite_edge_interior_conflict_2

A predicate object type. Must provide *bool operator()(Site_2 s1, Site_2 s2, Site_2 s3, Site_2 s4, Site_2 q, Sign sgn)*. The sites *s1*, *s2*, *s3* and *s4* define a Voronoi edge that lies on the bisector of *s1* and *s2* and has as endpoints the Voronoi vertices defined by the triplets *s1*, *s2*, *s3* and *s1*, *s4* and *s2*. The sign *sgn* is the common sign of the distance of the site q from the Voronoi circle of the triplets *s1*, *s2*, *s3* and *s1*, *s4* and *s2*. In case that *sgn* is equal to *NEGATIVE*, the predicate returns *true* if and only if the entire Voronoi edge is in conflict with q. If *sgn* is equal to *POSITIVE* or *ZERO*, the predicate returns *false* if and only if q is not in conflict with the Voronoi edge.
Precondition: the Voronoi vertices of *s1*, *s2*, *s3*, and *s1*, *s4*, *s2* must exist.
Must also provide *bool operator()(Site_2 s1, Site_2 s2, Site_2 s3, Site_2 q, Sign sgn)*. The sites *s1*, *s2*, *s3* and the site at infinity s_∞ define a Voronoi edge that lies on the bisector of *s1* and *s2* and has as endpoints the Voronoi vertices v_{123} and $v_{1\infty2}$ defined by the triplets *s1*, *s2*, *s3* and *s1*, s_∞ and *s2* (the second vertex is actually at infinity). The sign *sgn* is the common sign of the distance of the site q from the two Voronoi circles centered at the Voronoi vertices v_{123} and $v_{1\infty2}$. In case that *sgn* is *NEGATIVE*, the predicate returns *true* if and only if the entire Voronoi edge is in conflict with q. If *sgn* is *POSITIVE* or *ZERO*, the predicate returns *false* if and only if q is not in conflict with the Voronoi edge.
Precondition: the Voronoi vertex v_{123} of *s1*, *s2*, *s3* must exist.
Must finally provide *bool operator()(Site_2 s1, Site_2 s2, Site_2 q, Sign sgn)*. The sites *s1*, *s2* and the site at infinity s_∞ define a Voronoi edge that lies on the bisector of $v_{12\infty}$ and $v_{1\infty2}$ *s1* and *s2* and has as endpoints the Voronoi vertices defined by the triplets *s1*, *s2*, s_∞ and *s1*, s_∞ and *s2* (both vertices are actually at infinity). The sign *sgn* denotes the common sign of the distance of the site q from the Voronoi circles centered at $v_{12\infty}$ and $v_{1\infty2}$. If *sgn* is *NEGATIVE*, the predicate returns *true* if and only if the entire Voronoi edge is in conflict with q. If *POSITIVE* or *ZERO* is *false*, the predicate returns *false* if and only if q is not in conflict with the Voronoi edge.

SegmentDelaunayGraphTraits_2:: Infinite_edge_interior_conflict_2

A predicate object type. Must provide *bool operator()(Site_2 s1, Site_2 s2, Site_2 s3, Site_2 q, Sign sgn)*. The sites s_∞, *s1*, *s2* and *s3* define a Voronoi edge that lies on the bisector of s_∞ and *s1* and has as endpoints the Voronoi vertices $v_{\infty12}$ and $v_{\infty31}$ defined by the triplets s_∞, *s1*, *s2* and s_∞, *s3* and *s1*. The sign *sgn* is the common sign of the distances of q from the Voronoi circles centered at the vertices $v_{\infty12}$ and $v_{\infty31}$. If *sgn* is *NEGATIVE*, the predicate returns *true* if and only if the entire Voronoi edge is in conflict with q. If *sgn* is *POSITIVE* or *ZERO*, the predicate returns *false* if and only if q is not in conflict with the Voronoi edge.

SegmentDelaunayGraphTraits_2:: Oriented_side_2

A predicate object type. Must provide *Oriented_side operator()(Site_1 s1, Site_2 s2, Site_2 s3, Site_2 s, Site_2 p)*. Determines the oriented side of the line ℓ that contains the point site p, where ℓ is the line that passes through the Voronoi vertex of the sites *s1*, *s2*, *s3* and is perpendicular to the segment site s.
Precondition: s must be a segment and p must be a point.

SegmentDelaunayGraphTraits_2:: Arrangement_type_2

A predicate object type. Must provide *Arrangement_type operator()(Site_2 s1, Site_2 s2)* that returns the type of the arrangement of the two sites *s1* and *s2*.

Access to predicate objects

Compare_x_2	*gt.compare_x_2_object()*
Compare_y_2	*gt.compare_y_2_object()*
Orientation_2	*gt.orientation_2_object()*
Equal_2	*gt.equal_2_object()*
Are_parallel_2	*gt.are_parallel_2_object()*
Oriented_side_of_bisector_2	*gt.oriented_side_of_bisector_test_2_object()*
Vertex_conflict_2	*gt.vertex_conflict_2_object()*
Finite_edge_interior_conflict_2	*gt.finite_edge_interior_conflict_2_object()*
Infinite_edge_interior_conflict_2	*gt.infinite_edge_interior_conflict_2_object()*
Oriented_side_2	*gt.oriented_side_2_object()*
Arrangement_type_2	*gt.arrangement_type_2_object()*

Access to contructor objects

Construct_object_2	*gt.construct_object_2_object()*
Construct_svd_vertex_2	*gt.construct_svd_vertex_2_object()*

Access to other objects

Assign_2	*gt.assign_2_object()*

Has Models

CGAL::Segment_Delaunay_graph_traits_2<K,MTag>
CGAL::Segment_Delaunay_graph_traits_without_intersections_2<K,MTag>
CGAL::Segment_Delaunay_graph_filtered_traits_2<CK,CM,EK,EM,FK,FM>
CGAL::Segment_Delaunay_graph_filtered_traits_without_intersections_2<CK,CM,EK,EM,FK,FM>

See Also

SegmentDelaunayGraphSite_2
CGAL::Segment_Delaunay_graph_2<Gt,DS>

CGAL::Segment_Delaunay_graph_hierarchy_2<Gt,STag,DS>
CGAL::Segment_Delaunay_graph_traits_2<K,MTag>
CGAL::Segment_Delaunay_graph_traits_without_intersections_2<K,MTag>
CGAL::Segment_Delaunay_graph_filtered_traits_2<CK,CM,EK,EM,FK,FM>
CGAL::Segment_Delaunay_graph_filtered_traits_without_intersections_2<CK,CM,EK,EM,FK,FM>

Class

CGAL::Segment_Delaunay_graph_traits_2<K,MTag>

Definition

The class *Segment_Delaunay_graph_traits_2<K,MTag>* provides a model for the *SegmentDelaunayGraphTraits_2* concept. This class has two template parameters. The first template parameter must be a model of the *Kernel* concept. The second template parameter corresponds to how predicates are evaluated. There are two possible values for *MTag*, namely *CGAL::Field_with_sqrt_tag* and *CGAL::Field_tag*. The first one must be used when the number type used in the representation supports the exact evaluation of signs of expressions involving all four basic operations and square roots, whereas the second one requires the exact evaluation of signs of field-type expressions, i.e., expressions involving additions, subtractions, multiplications and divisions. The default value for *MTag* is *CGAL::Field_tag*. The way the predicates are evaluated is discussed in [Bur96] and [Kar04] (the geometric filtering part).

#include <CGAL/Segment_Delaunay_graph_traits_2.h>

Is Model for the Concepts

SegmentDelaunayGraphTraits_2

Types

typedef CGAL::Tag_true	*Intersections_tag;*

The *Segment_Delaunay_graph_traits_2<K,MTag>* class introduces a few additional types with respect to the *SegmentDelaunayGraphTraits_2* concept. These are:

typedef K	*Kernel;*	A typedef for the template parameter *K*.
typedef MTag	*Method_tag;*	A typedef for the template parameter *MTag*.

See Also

Kernel
SegmentDelaunayGraphTraits_2
CGAL::Field_tag
CGAL::Field_with_sqrt_tag
CGAL::Segment_Delaunay_graph_2<Gt,DS>
CGAL::Segment_Delaunay_graph_hierarchy_2<Gt,STag,DS>
CGAL::Segment_Delaunay_graph_traits_without_intersections_2<K,MTag>
CGAL::Segment_Delaunay_graph_filtered_traits_2<CK,CM,EK,EM,FK,FM>
CGAL::Segment_Delaunay_graph_filtered_traits_without_intersections_2<CK,CM,EK,EM,FK,FM>

CGAL::Segment_Delaunay_graph_traits_without_intersections_2< K,MTag>

Class

Definition

The class *Segment_Delaunay_graph_traits_without_intersections_2<K,MTag>* provides a model for the *SegmentDelaunayGraphTraits_2* concept. This class has two template parameters. The first template parameter must be a model of the *Kernel* concept. The second template parameter corresponds to how predicates are evaluated. There are two possible values for *MTag*, namely *CGAL::Field_with_sqrt_tag* and *CGAL::Euclidean_ring_tag*. The first one must be used when the number type used in the representation supports the exact evaluation of signs of expressions involving all four basic operations and square roots, whereas the second one requires the exact evaluation of signs of ring-type expressions, i.e., expressions involving only additions, subtractions and multiplications. The default value for *MTag* is *CGAL::Euclidean_ring_tag*. The way the predicates are evaluated is discussed in [Bur96] and [Kar04] (the geometric filtering part).

#include <CGAL/Segment_Delaunay_graph_traits_2.h>

Is Model for the Concepts

SegmentDelaunayGraphTraits_2

Types

typedef CGAL::Tag_false *Intersections_tag;*

The *Segment_Delaunay_graph_traits_without_intersections_2<K,MTag>* class introduces a few additional types with respect to the *SegmentDelaunayGraphTraits_2* concept. These are:

typedef K	*Kernel;*	A typedef for the template parameter *K*.
typedef MTag	*Method_tag;*	A typedef for the template parameter *MTag*.

See Also

Kernel
SegmentDelaunayGraphTraits_2
CGAL::Euclidean_ring_tag
CGAL::Field_with_sqrt_tag
CGAL::Segment_Delaunay_graph_2<Gt,DS>
CGAL::Segment_Delaunay_graph_hierarchy_2<Gt,STag,DS>
CGAL::Segment_Delaunay_graph_traits_2<K,MTag>
CGAL::Segment_Delaunay_graph_filtered_traits_2<CK,CM,EK,EM,FK,FM>
CGAL::Segment_Delaunay_graph_filtered_traits_without_intersections_2<CK,CM,EK,EM,FK,FM>

Class

CGAL::Segment_Delaunay_graph_filtered_traits_2< CK,CM,EK,EM,FK,FM>

Definition

The class *Segment_Delaunay_graph_filtered_traits_2<CK,CM,EK,EM,FK,FM>* provides a model for the *SegmentDelaunayGraphTraits_2* concept.

The class *Segment_Delaunay_graph_filtered_traits_2<CK,CM,EK,EM,FK,FM>* uses the filtering technique [BBP01] to achieve traits for the *Segment_Delaunay_graph_2<Gt,DS>* class with efficient and exact predicates given an exact kernel *EK* and a filtering kernel *FK*. The geometric constructions associated provided by this class are equivalent to those provided by the traits class *Segment_Delaunay_graph_traits_2<CK,CM>*, which means that they may be inexact depending on the choice of the *CK* kernel.

This class has six template parameters. The first, third and fifth template parameters must be a models of the *Kernel* concept. The parameter *CK* is the construction kernel and it is the kernel that will be used for constructions. The parameter *FK* is the filtering kernel; this kernel will be used for performing the arithmetic filtering for the predicates involved in the computation of the segment Delaunay graph. Finally, the parameter *EK* is the exact kernel; this kernel will be used for computing the predicates if the filtering kernel fails to produce an answer.

The second, fourth and sixth template parameters correspond to how predicates are evaluated. There are two predefined possible values for these parameters, namely *CGAL::Field_with_sqrt_tag* and *CGAL::Field_tag*. The first one must be used when the number type used in the representation supports the exact evaluation of signs of expressions involving all four basic operations and square roots, whereas the second one requires that only field operations are exact. Finally, in order to get exact constructions *CM* must be set to *CGAL::Field_with_sqrt_tag* and the number type in *CK* must support operations involing divisions and square roots (as well as the other three basic operations of course). The way the predicates are evaluated is discussed in [Bur96] and [Kar04] (the geometric filtering part).

The default values for the template parameters are as follows: *CM = CGAL::Field_with_sqrt_tag* (it is assumed that *CGAL::Cartesian<double>* or *CGAL::Simple_cartesian<double>* will be the entry for the template parameter *CK*), *EM = CGAL::Field_tag*, *FK = CGAL::Simple_cartesian<CGAL::Interval_nt<false> >*, *FM = CGAL::Field_with_sqrt_tag*. If the GMP package is installed with CGAL, the template parameter *EK* has the default value: *EK = CGAL::Simple_cartesian<CGAL::Gmpq>*, otherwise its default value is *EK = CGAL::Simple_cartesian<CGAL::Quotient<CGAL::MP_Float> >*.

#include <CGAL/Segment_Delaunay_graph_filtered_traits_2.h>

Is Model for the Concepts

SegmentDelaunayGraphTraits_2
DefaultConstructible
CopyConstructible
Assignable

Types

typedef CGAL::Tag_true   Intersections_tag;

In addition to the types required by the *SegmentDelaunayGraphTraits_2* concept the class *Segment_Delaunay_graph_filtered_traits_2<CK,CM,EK,EM,FK,FM>* defines the following types:

typedef CK	*Kernel;*
typedef CK	*Construction_kernel;*
typedef FK	*Filtering_kernel;*
typedef EK	*Exact_kernel;*
typedef CM	*Method_tag;*
typedef CM	*Construction_traits_method_tag;*
typedef FM	*Filtering_traits_method_tag;*
typedef EM	*Exact_traits_method_tag;*

Segment_Delaunay_graph_filtered_traits_2<CK,CM,EK,EM,FK,FM>:: Construction_traits

A type for the segment Delaunay graph traits, where the kernel is *CK*.

Segment_Delaunay_graph_filtered_traits_2<CK,CM,EK,EM,FK,FM>:: Filtering_traits

A type for the segment Delaunay graph traits, where the kernel is *FK*.

Segment_Delaunay_graph_filtered_traits_2<CK,CM,EK,EM,FK,FM>:: Exact_traits

A type for the segment Delaunay graph traits, where the kernel is *EK*.

See Also

Kernel
SegmentDelaunayGraphTraits_2
CGAL::Field_tag
CGAL::Field_with_sqrt_tag
CGAL::Segment_Delaunay_graph_2<Gt,DS>
CGAL::Segment_Delaunay_graph_hierarchy_2<Gt,STag,DS>
CGAL::Segment_Delaunay_graph_traits_2<K,MTag>
CGAL::Segment_Delaunay_graph_traits_without_intersections_2<K,MTag>
CGAL::Segment_Delaunay_graph_filtered_traits_without_intersections_2<CK,CM,EK,EM,FK,FM>

CGAL::Segment_Delaunay_graph_filtered_traits_without_intersections_2<CK,CM,EK,EM,FK,FM>

Definition

The class *Segment_Delaunay_graph_filtered_traits_without_intersections_2<CK,CM,EK,EM,FK,FM>* provides a model for the *SegmentDelaunayGraphTraits_2* concept.

The class *Segment_Delaunay_graph_filtered_traits_without_intersections_2<CK,CM,EK,EM,FK,FM>* uses the filtering technique [BBP01] to achieve traits for the *Segment_Delaunay_graph_2<Gt,DS>* class with efficient and exact predicates given an exact kernel *EK* and a filtering kernel *FK*. The geometric constructions associated provided by this class are equivalent to those provided by the traits class *Segment_Delaunay_graph_traits_without_intersections_2<CK,CM>*, which means that they may be inexact, depending on the choice of the *CK* kernel.

This class has six template parameters. The first, third and fifth template parameters must be a models of the *Kernel* concept. The parameter *CK* is the construction kernel and it is the kernel that will be used for constructions. The parameter *FK* is the filtering kernel; this kernel will be used for performing the arithmetic filtering for the predicates involved in the computation of the segment Delaunay graph. Finally, the parameter *EK* is the exact kernel; this kernel will be used for computing the predicates if the filtering kernel fails to produce an answer.

The second, fourth and sixth template parameters correspond to how predicates are evaluated. There are two predefined possible values for these parameters, namely *CGAL::Field_with_sqrt_tag* and *CGAL::Euclidean_ring_tag*. The first one must be used when the number type used in the representation supports the exact evaluation of signs of expressions involving all four basic operations and square roots, whereas the second requires the exact evaluation of signs of ring-type expressions, i.e., expressions involving only additions, subtractions and multiplications. Finally, in order to get exact constructions *CM* must be set to *CGAL::Field_with_sqrt_tag* and the number type in *CK* must support operations involing divisions and square roots (as well as the other three basic operations of course). The way the predicates are evaluated is discussed in [Bur96] and [Kar04] (the geometric filtering part).

The default values for the template parameters are as follows: *CM = CGAL::Field_with_sqrt_tag* (it is assumed that *CGAL::Cartesian<double>* or *CGAL::Simple_cartesian<double>* will be the entry for the template parameter *CK*), *EM = CGAL::Euclidean_ring_tag*, *FK = CGAL::Simple_cartesian<CGAL::Interval_nt<false> >*, *FM = CGAL::Field_with_sqrt_tag*. If the GMP package is installed with CGAL, the template parameter *EK* has the default value: *EK = CGAL::Simple_cartesian<CGAL::Gmpq>*, otherwise its default value is *EK = CGAL::Simple_cartesian<CGAL::MP_Float>*.

#include <CGAL/Segment_Delaunay_graph_filtered_traits_2.h>

Is Model for the Concepts

SegmentDelaunayGraphTraits_2
DefaultConstructible
CopyConstructible
Assignable

Types

typedef CGAL::Tag_false *Intersections_tag;*

In addition to the types required by the *SegmentDelaunayGraphTraits_2* concept the class *Segment_Delaunay_graph_filtered_traits_without_intersections_2<CK,CM,EK,EM,FK,FM>* defines the following types:

typedef CK	*Kernel;*
typedef CK	*Construction_kernel;*
typedef FK	*Filtering_kernel;*
typedef EK	*Exact_kernel;*
typedef CM	*Method_tag;*
typedef CM	*Construction_traits_method_tag;*
typedef FM	*Filtering_traits_method_tag;*
typedef EM	*Exact_traits_method_tag;*

Segment_Delaunay_graph_filtered_traits_without_intersections_2<CK,CM,EK,EM,FK,FM>:: Construction_traits

A type for the segment Delaunay graph traits, where the kernel is *CK*.

Segment_Delaunay_graph_filtered_traits_without_intersections_2<CK,CM,EK,EM,FK,FM>:: Filtering_traits

A type for the segment Delaunay graph traits, where the kernel is *FK*.

Segment_Delaunay_graph_filtered_traits_without_intersections_2<CK,CM,EK,EM,FK,FM>:: Exact_traits

A type for the segment Delaunay graph traits, where the kernel is *EK*.

See Also

Kernel
SegmentDelaunayGraphTraits_2
CGAL::Euclidean_ring_tag
CGAL::Field_with_sqrt_tag
CGAL::Segment_Delaunay_graph_2<Gt,DS>
CGAL::Segment_Delaunay_graph_hierarchy_2<Gt,STag,DS>
CGAL::Segment_Delaunay_graph_traits_2<K,MTag>
CGAL::Segment_Delaunay_graph_traits_without_intersections_2<K,MTag>
CGAL::Segment_Delaunay_graph_filtered_traits_2<CK,CM,EK,EM,FK,FM>

CGAL::Segment_Delaunay_graph_hierarchy_2<Gt,STag,DS>

Definition

We provide an alternative to the class *Segment_Delaunay_graph_2<Gt,DS>* for the incremental construction of the segment Delaunay graph. The *Segment_Delaunay_graph_hierarchy_2<Gt,STag,DS>* class maintains a hierarchy of Delaunay graphs. There are two possibilities as to how this hierarchy is constructed.

In the first case the bottom-most level of the hierarchy contains the full segment Delaunay graph. The upper levels of the hierarchy contain only points that are either point sites or endpoints of segment sites in the bottom-most Delaunay graph. A point that is in level i (either as an individdual point or as the endpoint of a segment), is inserted in level $i+1$ with probability $1/\alpha$ where $\alpha > 1$ is some constant. In the second case the upper levels of the hierarchy contains not only points but also segments. A site that is in level i, is in level $i+1$ with probability $1/\beta$ where $\beta > 1$ is some constant.

The difference between the *Segment_Delaunay_graph_2<Gt,DS>* class and the *Segment_Delaunay_graph_hierarchy_2<Gt,STag,DS>* class (both versions of it) is on how the nearest neighbor location is done. Given a point p the location is done as follows: at the top most level we find the nearest neighbor of p as in the *Segment_Delaunay_graph_2<Gt,DS>* class. At every subsequent level i we use the nearest neighbor found at level $i+1$ to find the nearest neighbor at level i. This is a variant of the corresponding hierarchy for points found in [Dev02]. The details are described in [Kar04].

The class has three template parameters. The first and third have essentially the same semantics as in the *Segment_Delaunay_graph_2<Gt,DS>* class. The first template parameter must be a model of the *SegmentDelaunayGraphTraits_2* concept. The third template parameter must be a model of the *SegmentDelaunayGraphDataStructure_2* concept. However, the vertex base class that is to be used in the segment Delaunay graph data structure must be a model of the *SegmentDelaunayGraphHierarchyVertexBase_2* concept. The third template parameter defaults to *Triangulation_data_structure_2< Segment_Delaunay_graph_hierarchy_vertex_base_2< Segment_Delaunay_graph_vertex_base_2<Gt> >, Triangulation_face_base_2<Gt> >*. The second template parameter controls whether or not segments are added in the upper levels of the hierarchy. It's possible values are *CGAL::Tag_true* and *CGAL::Tag_false*. If it is set to *CGAL::Tag_true*, segments are also inserted in the upper levels of the hierarchy. The value *CGAL::Tag_false* indicates that only points are to be inserted in the upper levels of the hierarchy. The default value for the second template parameter is *CGAL::Tag_false*.

The *Segment_Delaunay_graph_hierarchy_2<Gt,STag,DS>* class derives publicly from the *Segment_Delaunay_graph_2<Gt,DS>* class. The interface is the same with its base class. In the sequel only additional types and methods defined are documented.

#include <CGAL/Segment_Delaunay_graph_hierarchy_2.h>

Is Model for the Concepts

DefaultConstructible
CopyConstructible
Assignable

Inherits From

CGAL::Segment_Delaunay_graph_2<Gt,DS>

Types

Segment_Delaunay_graph_hierarchy_2<Gt,STag,DS> introduces the following types in addition to those introduced by its base class *Segment_Delaunay_graph_2<Gt,DS>*.

typedef STag	*Segments_in_hierarchy_tag;*	A type for the *STag* template parameter.
typedef CGAL::Segment_Delaunay_graph_2<Gt,DS>	*Base;*	A type for the base class.

Creation

In addition to the default and copy constructors, the following constructors are defined:

Segment_Delaunay_graph_hierarchy_2<Gt,STag,DS> sdgh(Gt gt=Gt());

Creates a hierarchy of segment Delaunay graphs using *gt* as geometric traits.

template< class Input_iterator >
Segment_Delaunay_graph_hierarchy_2<Gt,STag,DS> sdgh(Input_iterator first, Input_iterator beyond, Gt gt=Gt())

Creates a segment Delaunay graph hierarchy using *gt* as geometric traits and inserts all sites in the range [*first*, *beyond*). *Input_iterator* must be a model of *InputIterator*. The value type of *Input_iterator* must be either *Point_2* or *Site_2*.

I/O

std::ostream&	*std::ostream& os << svdh*	Writes the current state of the segment Delaunay graph hierarchy to an output stream. In particular, all sites in the diagram are written to the stream (represented through appropriate input sites), as well as the underlying combinatorial hierarchical data structure.
std::istream&	*std::istream& is >> svdh*	Reads the state of the segment Delaunay graph hierarchy from an input stream.

See Also

SegmentDelaunayGraphDataStructure_2
SegmentDelaunayGraphTraits_2
SegmentDelaunayGraphHierarchyVertexBase_2
CGAL::Segment_Delaunay_graph_2<Gt,DS>
CGAL::Triangulation_data_structure_2<Vb,Fb>
CGAL::Segment_Delaunay_graph_traits_2<K,MTag>
CGAL::Segment_Delaunay_graph_traits_without_intersections_2<K,MTag>
CGAL::Segment_Delaunay_graph_filtered_traits_2<CK,CM,EK,EM,FK,FM>
CGAL::Segment_Delaunay_graph_filtered_traits_without_intersections_2<CK,CM,EK,EM,FK,FM>
CGAL::Segment_Delaunay_graph_hierarchy_vertex_base_2<Vbb>

SegmentDelaunayGraphHierarchyVertexBase_2

Definition

The vertex of a segment Delaunay graph included in a segment Delaunay graph hierarchy has to provide some pointers to the corresponding vertices in the graphs of the next and preceeding levels. Therefore, the concept *SegmentDelaunayGraphHierarchyVertexBase_2* refines the concept *SegmentDelaunayGraphVertexBase_2*, by adding two vertex handles to the correponding vertices for the next and previous level graphs.

Refines

SegmentDelaunayGraphVertexBase_2

Types

SegmentDelaunayGraphHierarchyVertexBase_2 does not introduce any types in addition to those of *SegmentDelaunayGraphVertexBase_2*.

Creation

The *SegmentDelaunayGraphHierarchyVertexBase_2* concept does not introduce any constructors in addition to those of the *SegmentDelaunayGraphVertexBase_2* concept.

Operations

Vertex_handle	*v.up()*	Returns a handle to the corresponding vertex of the next level segment Delaunay graph. If such a vertex does not exist *Vertex_handle()* is returned.
Vertex_handle	*v.down()*	Returns a handle to the corresponding vertex of the previous level segment Delaunay graph. If such a vertex does not exist *Vertex_handle()* is returned.
void	*v.set_up(Vertex_handle u)*	Sets the handle for the vertex of the next level segment Delaunay graph.
void	*v.set_down(Vertex_handle d)*	Sets the handle for the vertex of the previous level segment Delaunay graph.

Has Models

CGAL::Segment_Delaunay_graph_hierarchy_vertex_base_2<CGAL::Segment_Delaunay_graph_vertex_base_2<Gt,SSTag> >

See Also

SegmentDelaunayGraphDataStructure_2
SegmentDelaunayGraphVertexBase_2
CGAL::Segment_Delaunay_graph_hierarchy_2<Gt,DS>
CGAL::Triangulation_data_structure_2<Vb,Fb>
CGAL::Segment_Delaunay_graph_vertex_base_2<Gt,SSTag>
CGAL::Segment_Delaunay_graph_hierarchy_vertex_base_2<Vbb>

CGAL::Segment_Delaunay_graph_hierarchy_vertex_base_2<Vbb>

Class

Definition

The class *Segment_Delaunay_graph_hierarchy_vertex_base_2<Vbb>* provides a model for the *SegmentDelaunayGraphHierarchyVertexBase_2* concept, which is the vertex base required by the *Segment_Delaunay_graph_hierarchy_2<Gt,DS>* class. The class *Segment_Delaunay_graph_hierarchy_vertex_base_2<Vbb>* is templated by a class *Vbb* which must be a model of the *SegmentDelaunayGraphVertexBase_2* concept.

#include <CGAL/Segment_Delaunay_graph_hierarchy_vertex_base_2.h>

Inherits From

Vbb

Is Model for the Concepts

SegmentDelaunayGraphHierarchyVertexBase_2

See Also

SegmentDelaunayGraphVertexBase_2
SegmentDelaunayGraphHierarchyVertexBase_2
SegmentDelaunayGraphDataStructure_2
CGAL::Segment_Delaunay_graph_vertex_base_2<Gt,SSTag>
CGAL::Triangulation_data_structure_2<Vb,Fb>
CGAL::Segment_Delaunay_graph_hierarchy_2<Gt,STag,DS>

Chapter 45

2D Apollonius Graphs (Delaunay Graphs of Disks)

Menelaos Karavelas and Mariette Yvinec

Contents

This chapter describes the two-dimensional Apollonius graph of CGAL. We start with a few definitions in Section 45.1. The software design of the 2D Apollonius graph package is described in Section 45.2. In Section 45.3 we discuss the geometric traits of the 2D Apollonius graph package and in Section 45.4 the Apollonius graph hierarchy, a data structure suitable for fast nearest neighbor queries, is briefly described.

45.1 Definitions

The 2D Apollonius graph class of CGAL is designed to compute the dual of the *Apollonius diagram* or, as it is also known, the *Additively weighted Voronoi diagram*. The algorithm that has been implemented is dynamic, which means that we can perform insertions and deletions on line. The corresponding CGAL class is called *Apollonius_graph_2<ApolloniusGraphTraits_2,ApolloniusGraphDataStructure_2>* and will be discussed in more detail in the sequel. The interested reader may want to refer to the paper by Karavelas and Yvinec [KY02] for the general idea as well as the details of the algorithm implemented.

Before describing the details of the implementation we make a brief introduction to the theory of Apollonius diagrams. The Apollonius diagram is defined over a set of sites $P_i = (c_i, w_i)$, $i = 1, \ldots, n$, where c_i is the point

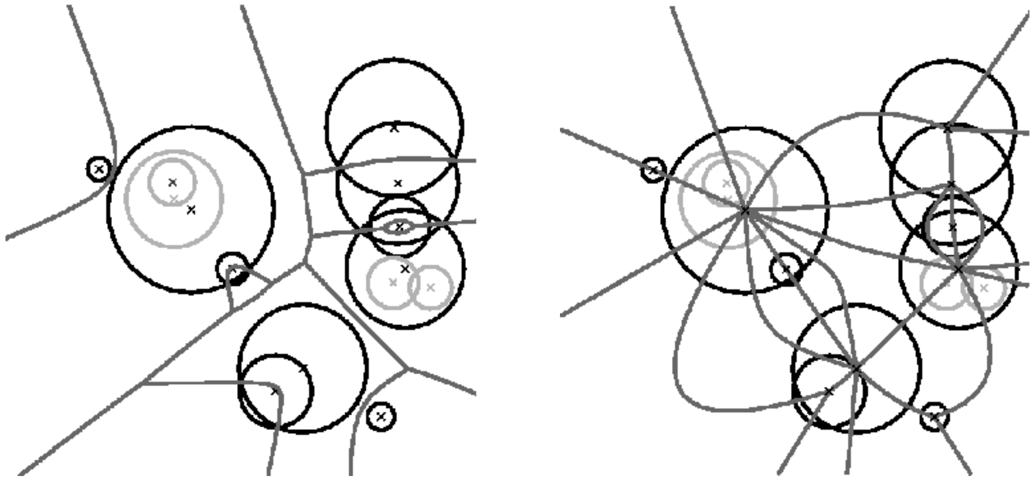


Figure 45.1: The Apollonius diagram (left) and its dual the Apollonius graph (right).

and w_i the weight of P_i. It is a subdivision of the plane into connected regions, called *cells*, associated with the sites (see Fig. 45.1(left)). The cell of a site P_i is the locus of points on the plane that are closer to P_i than any other site P_j, $j \neq i$. The distance $\delta(x, P_i)$ of a point x in the plane to a site P_i is defined as:

$$\delta(x, P_i) = \|x - c_i\| - w_i,$$

where $\|\cdot\|$ denotes the Euclidean norm. It can easily be seen that it is a generalization of the Voronoi diagram for points, which can actually be obtained if all the weights w_i are equal. Unlike the case of points, however, it is possible that a site P_i might have an empty cell. This can also happen in the case of the power diagram, whose dual is the regular triangulation (see Section 37.6). If this is the case we call the site *hidden* (these are the black circles in Fig. 45.1). A site which is not hidden will be referred to as *visible*.

If all weights w_i are non-negative, the Apollonius diagram can be viewed as the Voronoi diagram of the set of circles $\{P_1, \ldots, P_n\}$, where c_i is the center of the circle P_i and w_i its radius. If the weights are allowed to be negative, we need to go to 3D in order to explain what the Apollonius diagram means geometrically. We identify the 2D Euclidean plane with the xy-plane in 3D. Then the Voronoi diagram of a set of points can be seen as the vertical projection on the xy-plane of the lower envelope of a set of 3D cones defined as follows: for each point p in the set of 2D points we have a cone C_p whose apex is the point p. The axis of C_p is a line parallel to the z-axis passing through p, the angle of C_p is $45°$ and, finally C_p is facing in the positive z-direction (that is, C_p is contained in the positive z-halfspace). The Apollonius diagram corresponds to shifting the apexes of these cones in the z-direction by a quantity equal to the weight. Sites with negative weight will give rise to cones whose apex is in the negative z-halfspace and sites with positive weight will give rise to cones whose apex is in the positive z-halfspace. In a manner analogous to the case of points, the Apollonius diagram can then be defined as the vertical projection on the xy-plane of the lower envelope of the set of shifted cones. Notice that when all apexes are translated along the z-direction by the same amount, the projection of the lower envelope of the set of cones does not change. In particular, we can translate all cones by a large enough amount so that all apexes are in the positive z-halfspace. Algebraically, this means that the Apollonius diagram does not change if we add to all weights the same quantity, which in particular, implies that we can assume without loss of generality that all weights are positive. Given the observations above and in order to simplify our discussion of Apollonius diagrams, we will, from now on, assume that all weights are positive, and we will refer to the sites as circles.

The Apollonius diagram is a planar graph, and so is its dual, the Apollonius graph. There are many ways to embed it on the plane and one such way is shown in Fig. 45.1(right). The Apollonius graph is uniquely defined once we have the Apollonius diagram. If the circles are in *general position* (see precise definition below), then the Apollonius graph is a graph with triangular faces away from the convex hull of the set of circles (by triangular we mean that every face has exactly three edges). Near the convex hull we may have some spikes (i.e., vertices of degree 1). To unify our approach and handling of the Apollonius graph we add to the set of (finite) circles a fictitious circle at infinity, which we call the *site at infinity*. We can then connect all vertices of the outer face of the Apollonius graph to the site at infinity which gives us a graph with the property that all of its faces are now triangular. However, the Apollonius graph is not a triangulation for two main reasons: we cannot always embed it on the plane with straight line segments that yield a triangulation and, moreover, we may have two faces of the graph that have two edges in common, which is not allowed in a triangulation. Both of these particularities appear when we consider the Apollonius graph of the set of circles in Fig. 45.1(right).

We would like to finish our brief introduction to the theory of Apollonius graphs by discussing the concept of general position. We say that a set of circles is in general position if no two triplets of circles have the same tritangent circle. This statement is rather technical and it is best understood in the context of points. The equivalent statement for points is that we have no two triplets of points that define the same circumcircle, or equivalently that no four points are co-circular. The statement about general position made above is a direct generalization of the (much simpler to understand) statement about points. On the contrary, when we have circles in degenerate position, the Apollonius graph has faces with more than three edges on their boundary. We can get a triangulated version of the graph by simply *triangulating* the corresponding faces in an arbitrary way. In fact the algorithm that has been implemented in CGAL has the property that it always returns a valid *triangulated* version of the Apollonius graph. By valid we mean that it contains the actual Apollonius graph (i.e., the actual dual of the Apollonius diagram) and whenever there are faces with more than three faces then they are triangulated. The way that they are triangulated depends on the order of insertion and deletion of the circles in the diagram.

One final point has to be made about hidden circles. First of all we would like to be more precise about our definition of hidden circles: we say that a circle is hidden if its cell has empty interior. This definition allows us to guarantee that all visible circles have cells that are two-dimensional regions. Geometrically the fact that a circle is hidden means that it is contained in the closure of the disk of another circle (see again Fig. 45.1). Note that a circle contained in the union of several disks, but not in the closure of any one of them, is not hidden.

Hidden circles pose an additional difficulty to our algorithm and software design. Since we allow circles to be inserted and deleted at wish, it is possible that a circle that was hidden at some point in time, may become visible at a later point in time; for example this can happen if we delete the circle that hides it. For this purpose we store hidden circles and have them reappear when they become visible. We will discuss this issue in detail below. For the time being it suffices to say that the user has the ability to control this behavior. More specifically it is possible to discard the circles that become hidden. This choice is totally natural when for example we expect to do only insertions, since in this case a circle that becomes hidden will never reappear. On the other hand if deletions are expected as well, then we lose the ability to have the hidden circles reappear.

Degenerate Dimensions. The dimension of the Apollonius graph is in general 2. The exceptions to this rule are as follows:

- The dimension is -1 if the Apollonius graph contains no circles.
- The dimension is 0 if the Apollonius graph contains exactly one visible circle.
- The dimension is 1 is the Apollonius graph contains exactly two visible circles.

45.2 Software Design

The 2D Apollonius graph class *Apollonius_graph_2<ApolloniusGraphTraits_2,ApolloniusGraphDataStructure_2>* follows the design of the triangulation package of CGAL. It is parametrized by two arguments:

- the **geometric traits** class. It provides the basic geometric objects involved in the algorithm, such as sites, points etc. It also provides the geometric predicates for the computation of the Apollonius graph, as well as some basic constructions that can be used, for example, to visualize the Apollonius graph or the Apollonius diagram. The geometric traits for the Apollonius graph will be discussed in more detail in the next section.
- the **Apollonius graph data structure**. This is essentially the same as the triangulation data structure (discussed in Chapter 38), augmented with some additional operations that are specific to Apollonius graphs. The corresponding concept is that of *ApolloniusGraphDataStructure_2*, which in fact is a refinement of the *TriangulationDataStructure_2* concept. The class *Triangulation_data_structure_2<Vb,Fb>* is a model of the concept *ApolloniusGraphDataStructure_2*. A default value for the corresponding template parameter is provided, so the user does not need to specify it.

Storing Hidden Sites. As we have already mentioned a circle is hidden if it is contained inside some visible circle. This creates a parent-child relationship between visible and hidden circles: the parent of a hidden circle is the visible circle that contains it. If more than one visible circles contain a hidden circle then the hidden circle can be assigned to any of the visible circles arbitrarily.

To store hidden circles we assign to every visible circle a list. This list comprises the hidden circles that are contained in the visible circle. The user can access the hidden circles associated with a visible circle through an iterator called *Hidden_sites_iterator*. This iterator is defined in the *ApolloniusGraphVertexBase_2* concept and is implemented by its model, the *Apollonius_graph_vertex_base_2<Gt,StoreHidden>* class. It is also possible to iterate through the entire set of hidden sites using an homonymous iterator defined by the *Apollonius_graph_2< Gt,Agds>* class.

Since storing hidden sites may not be of interest in some cases (e.g., for example this is the case if we only perform insertions in the Apollonius graph), the user has the possibility of controlling this behavior. More precisely, the class *Apollonius_graph_vertex_base_2<Gt,StoreHidden>* has two template parameters, the second of which is a Boolean value. This value is by default *true* and it indicates that hidden sites should be stored. The user can indicate that hidden sites may be discarded by setting this value to *false*.

45.3 The Geometric Traits

The predicates required for the computation of the Apollonius graph are rather complicated. It is not the purpose of this document to discuss them in detail. The interested reader may refer to the papers by Karavelas and Emiris for the details [KE02, KE03]. However, we would like to give a brief overview of what they compute. There are several predicates needed by this algorithm. We will discuss the most important/complicated ones. It turns out that it is much easier to describe them in terms of the Apollonius diagram, rather than the Apollonius graph. Whenever it is applicable we will also describe their meaning in terms of the Apollonius graph.

The first two geometric predicates are called *Is_hidden_2* and *Oriented_side_of_bisector_2*. The first one involves two circles, say P_1 and P_2. It determines if P_1 is hidden with respect to P_2; more precisely it checks whether the circle P_1 is contained in the closure of the disk defined by the circle P_2. As its name indicates, it determines if a circle is hidden or not. The second predicate involves two circles P_1 and P_2 and a point q. It answers the question whether q is closer to P_1 or P_2. Its name stems from the fact that answering the aforementioned question is equivalent to determining the oriented side of the bisector of P_1 and P_2 that contains the query point q. This predicate is used by the algorithm for closest neighbor queries for points.

The next geometric predicate is called *Vertex_conflict_2* and it involves four circles P_1, P_2, P_3, and P_4 (see Fig. 45.3). The first three (red circles in Fig. 45.3) define a tritangent circle (yellow circle in Fig. 45.3). What we want to determine is the sign of the distance of the green circle from the yellow circle. The distance between two circles $K_1 = (c_1, r_1)$ and $K_2 = (c_2, r_2)$ is defined as the distance of their centers minus their radii:

$$\delta(K_1, K_2) = \|c_1 - c_2\| - r_1 - r_2.$$

This predicate determines if a vertex in the Apollonius diagram (the center of the yellow circle) is destroyed when a new circle is inserted in the diagram (the green circle). In the Apollonius graph it tells us if a triangular face of the diagram is to be destroyed or not.

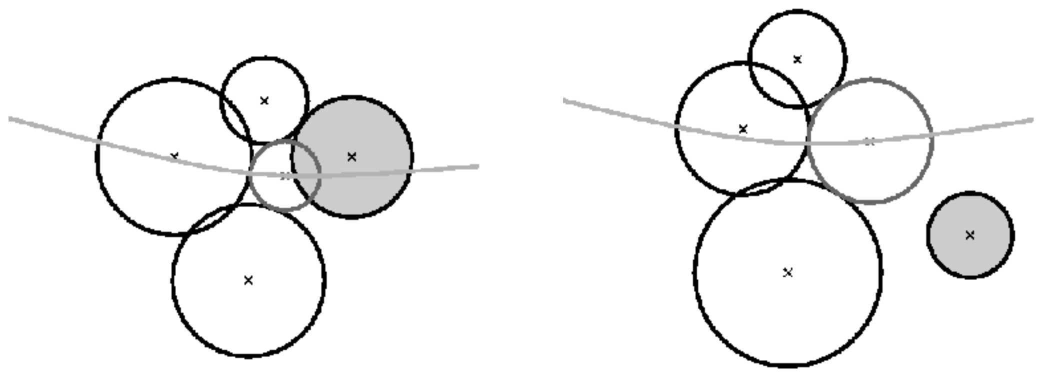


Figure 45.2: The *Vertex_conflict_2* predicate. The left-most, bottom-most and top-most circles define the tritangent circle in the middle. We want to determine the sign of the distance of the left-most circle from the one in the middle. The almost horizontal curve is the bisector of the top-most and bottom-most circles. Left: the predicate returns *CGAL::NEGATIVE*. Right: the predicate returns *CGAL::POSITIVE*.

What we essentially want to compute when we construct incrementally a Voronoi diagram, is whether the object to be inserted destroys an edge of the Voronoi diagram or not. In the case of points this is really easy and it amounts to the well known *incircle* test. In the case of circles the situation is more complicated. We can have six possible outcomes as to what portion of an edge of the Apollonius diagram the new circle destroys (see Fig. 45.3). The first two can be answered directly by the *Vertex_conflict_2* predicate evaluated for the two endpoints of the Apollonius diagram edge. This is due to the fact that the value of the *Vertex_conflict_2* predicate is different for the two endpoints. If the two values are the same then we need an additional test which determines if the interior of the Apollonius diagram edge is destroyed by the new circle. This is what the *Finite_edge_interior_conflict_2* and *Infinite_edge_interior_conflict_2* predicates do. In essence, it is the same predicate (same idea) applied to two different types of edges in the Apollonius diagram: a finite or an infinite edge. An edge is infinite if its dual edge in the Apollonius graph connects the site at infinity with the vertex corresponding to a (finite) circle; otherwise it is a finite edge.

The last predicate that we want to discuss is called *Is_degenerate_edge_2*. It tells us whether an edge in the Apollonius diagram is degenerate, that is if its two endpoints coincide. In the Apollonius graph such an edge corresponds to one of the additional edges that we use to triangulate the non-triangular faces.

The aforementioned predicates are part of the *ApolloniusGraphTraits_2* concept of CGAL. CGAL also provides a model for this concept, the *Apollonius_graph_traits_2<K,Method_tag>* class. The first template parameter of this class must be a model of the *Kernel* concept. The second template parameter is a tag that indicates what operations are allowed in the computations that take place within the traits class. The two possible values of the *Method_tag* parameter are *CGAL::Ring_tag* and *CGAL::Sqrt_field_tag*. When *CGAL::Ring_tag* is used, only ring operations are used during the evaluation of the predicates, whereas if *CGAL::Sqrt_field_tag* is chosen, all four field operations, as well as square roots, are used during the predicate evaluation.

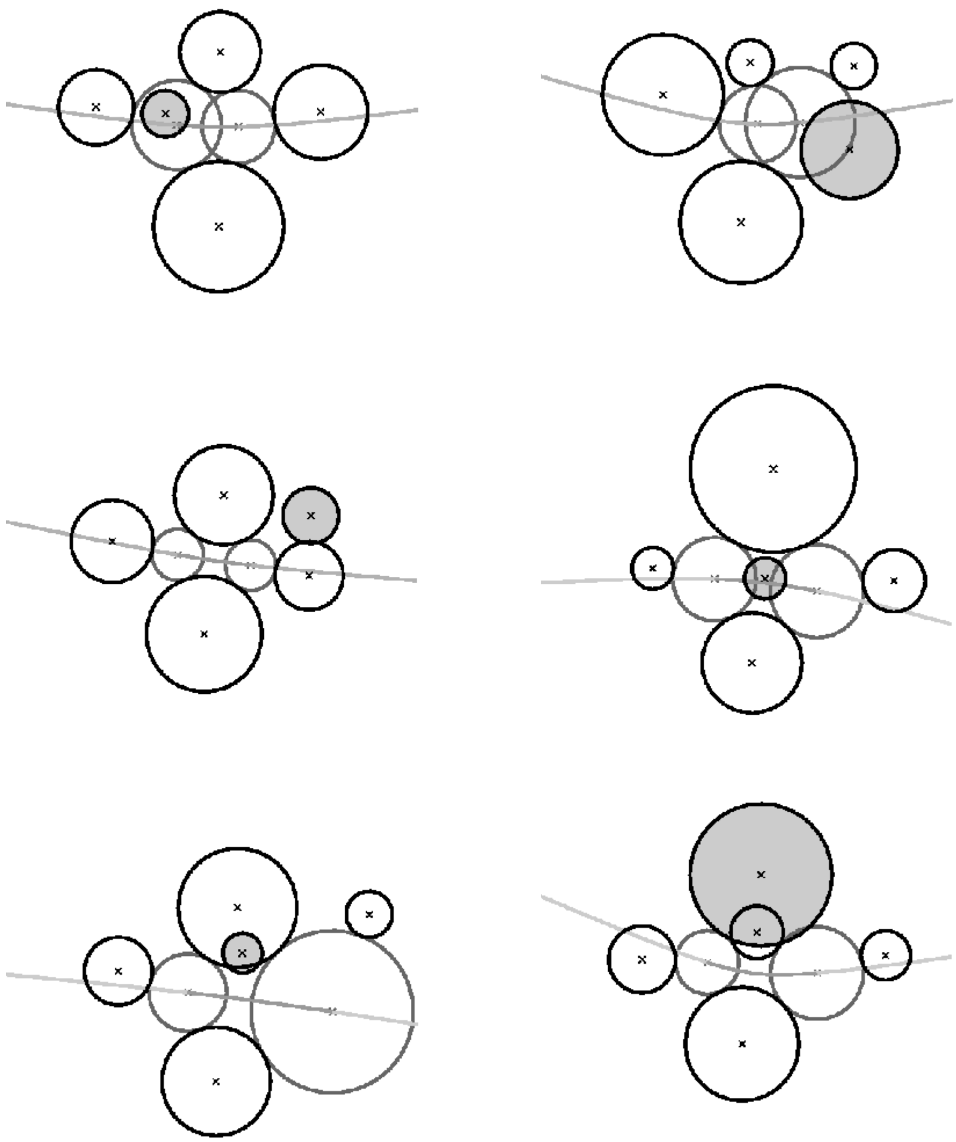


Figure 45.3: The 6 possible outcomes of the *Finite_edge_interior_conflict_2* predicate. Top left: only a neighborhood around the left-most endpoint of the edge will be destroyed. Top right: only a neighborhood around the right-most endpoint of the edge will be destroyed. Middle left: no portion of the edge is destroyed. Middle right: the entire edge will be destroyed. Bottom left: a neighborhood in the interior of the edge will be destroyed; the regions near the endpoints remain unaffected. Bottom right: The neighborhood around the two endpoints will be destroyed, but an interval in the interior of the edge will remain in the new diagram.

The *Apollonius_graph_traits_2<K,Method_tag>* class provides exact predicates if the number type in the kernel *K* is an exact number type. This is to be associated with the type of operations allowed for the predicate evaluation. For example *CGAL::MP_Float* as number type, with *CGAL::Ring_tag* as tag will give exact predicates, whereas *CGAL::MP_Float* with *CGAL::Sqrt_field_tag* will give inexact predicates.

Since using an exact number type may be too slow, the *Apollonius_graph_traits_2<K,Method_tag>* class is designed to support the dynamic filtering of CGAL through the *CGAL::Filtered_exact<CT,ET>* mechanism. In particular if *CT* is an inexact number type that supports the operations denoted by the tag *Method_tag* and *ET* is an exact number type for these operations, then kernel with number type *CGAL::Filtered_exact<CT,ET>* will yield exact predicates for the Apollonius graph traits. To give a concrete example, *CGAL::Filtered_exact< double,CGAL::MP_Float>* with *CGAL::Ring_tag* will produce exact predicates.

Another possibility for fast and exact predicate evaluation is to use the *Apollonius_graph_filtered_traits_2< CK,CM,EK,EM,FK,FM>* class. This class is the analog of a filtered kernel. It takes a constructions kernel *CK*, a filtering kernel *FK* and an exact kernel *EK*, as well as the corresponding tags (*CM*, *FM* and *EM*, respectively). It evaluates the predicates by first using the filtering kernel, and if this fails the evaluation is performed using the exact kernel. The constructions are done using the kernel *CK*, which means that they are not necessarily exact. All template parameters except *CK* have default values, which are explained in the reference manual.

45.4 The Apollonius Graph Hierarchy

The *Apollonius_graph_hierarchy_2<ApolloniusGraphTraits_2,ApolloniusGraphDataStructure_2>* class is nothing but the equivalent of the *Triangulation_hierarchy_2* class, applied to the Apollonius graph. It consists of a series of Apollonius graphs constructed in a manner analogous to the Delaunay hierarchy by Devillers [Dev98]. The class *Apollonius_graph_hierarchy_2<ApolloniusGraphTraits_2,ApolloniusGraphDataStructure_2>* has exactly the same interface and functionality as the *Apollonius_graph_2<ApolloniusGraphTraits_2,ApolloniusGraphDataStructure_2>* class. Using the Apollonius graph hierarchy involves an additional cost in space and time for maintaining the hierarchy. Our experiments have shown that it usually pays off to use the hierarchy for inputs consisting of more than 1,000 circles. This threshold holds for both the construction of the Apollonius diagram itself, as well as for nearest neighbor queries.

45.5 Examples

45.5.1 First Example

```
#include <iostream>
#include <fstream>
#include <cassert>

// the number type
#include <CGAL/MP_Float.h>


// example that uses an exact number type

typedef CGAL::MP_Float NT;

// choose the kernel
#include <CGAL/Simple_cartesian.h>

typedef CGAL::Simple_cartesian<NT>  Kernel;

// typedefs for the traits and the algorithm

#include <CGAL/Apollonius_graph_2.h>
```

```
#include <CGAL/Apollonius_graph_traits_2.h>

typedef CGAL::Apollonius_graph_traits_2<Kernel>   Traits;
typedef CGAL::Apollonius_graph_2<Traits>          Apollonius_graph;



int main()
{
  std::ifstream ifs("data/sites.cin");
  assert( ifs );

  Apollonius_graph ag;
  Apollonius_graph::Site_2 site;

  // read the sites and insert them in the Apollonius graph
  while ( ifs >> site ) {
    ag.insert(site);
  }

  // validate the Apollonius graph
  assert( ag.is_valid(true, 1) );
  std::cout << std::endl;

  return 0;
}
```

File: examples/Apollonius_graph_2/ag2_exact_traits.cpp

45.5.2 Second Example

```
#include <CGAL/basic.h>

// standard includes
#include <iostream>
#include <fstream>
#include <cassert>


#if defined CGAL_USE_LEDA
#  include <CGAL/leda_real.h>
#elif defined CGAL_USE_CORE
#  include <CGAL/CORE_Expr.h>
#endif


#if defined CGAL_USE_LEDA
// If LEDA is present use leda_real as the exact number type
typedef leda_real NT;

#elif defined CGAL_USE_CORE
// Otherwise if CORE is present use CORE's Expr as the exact number type
```

```
typedef CORE::Expr NT;

#else

// Otherwise just use double. This may cause numerical errors but it
// is still worth doing it to show how to define correctly the traits
// class
typedef double NT;

#endif


#include <CGAL/Simple_cartesian.h>

typedef CGAL::Simple_cartesian<NT>  Kernel;


// typedefs for the traits and the algorithm

#include <CGAL/Apollonius_graph_2.h>
#include <CGAL/Apollonius_graph_traits_2.h>

// the traits class is now going to assume that the operations
// +,-,*,/ and sqrt are supported exactly
typedef
CGAL::Apollonius_graph_traits_2<Kernel,CGAL::Field_with_sqrt_tag> Traits;

typedef CGAL::Apollonius_graph_2<Traits> Apollonius_graph;



int main()
{
  std::ifstream ifs("data/sites.cin");
  assert( ifs );

  Apollonius_graph ag;
  Apollonius_graph::Site_2 site;

  // read the sites and insert them in the Apollonius graph
  while ( ifs >> site ) {
    ag.insert(site);
  }

  // validate the Apollonius graph
  assert( ag.is_valid(true, 1) );
  std::cout << std::endl;

  return 0;
}
```

File: examples/Apollonius_graph_2/ag2_exact_traits_sqrt.cpp

45.5.3 Third Example

```
#include <iostream>
#include <fstream>
#include <cassert>

// example that uses the filtered traits

// choose the representation
#include <CGAL/Simple_cartesian.h>

typedef CGAL::Simple_cartesian<double> Rep;

#include <CGAL/Apollonius_graph_2.h>
#include <CGAL/Triangulation_data_structure_2.h>
#include <CGAL/Apollonius_graph_vertex_base_2.h>
#include <CGAL/Triangulation_face_base_2.h>
#include <CGAL/Apollonius_graph_filtered_traits_2.h>

// typedef for the traits; the filtered traits class is used
typedef CGAL::Apollonius_graph_filtered_traits_2<Rep> Traits;

// typedefs for the algorithm

// With the second template argument in the vertex base class being
// false, we indicate that there is no need to store the hidden sites.
// One case where this is indeed not needed is when we only do
// insertions, like in the main program below.
typedef CGAL::Apollonius_graph_vertex_base_2<Traits,false>   Vb;
typedef CGAL::Triangulation_face_base_2<Traits>              Fb;
typedef CGAL::Triangulation_data_structure_2<Vb,Fb>       Agds;
typedef CGAL::Apollonius_graph_2<Traits,Agds>    Apollonius_graph;


int main()
{
  std::ifstream ifs("data/sites.cin");
  assert( ifs );

  Apollonius_graph ag;
  Apollonius_graph::Site_2 site;

  // read the sites and insert them in the Apollonius graph
  while ( ifs >> site ) {
    ag.insert(site);
  }

  // validate the Apollonius graph
  assert( ag.is_valid(true, 1) );
  std::cout << std::endl;

  // now remove all sites
  std::cout << "Removing all sites... " << std::flush;
  while ( ag.number_of_vertices() > 0 ) {
```

```
    ag.remove( ag.finite_vertex() );
  }
  std::cout << "done!" << std::endl << std::endl;

  return 0;
}
```

File: examples/Apollonius_graph_2/ag2_filtered_traits_no_hidden.cpp

45.5.4 Fourth Example

```
#include <iostream>
#include <fstream>
#include <cassert>

// example that uses the filtered traits

#include <CGAL/MP_Float.h>
#include <CGAL/Simple_cartesian.h>

// constructions kernel (inexact)
typedef CGAL::Simple_cartesian<double> CK;

// exact kernel
typedef CGAL::Simple_cartesian<CGAL::MP_Float> EK;


// typedefs for the traits and the algorithm

#include <CGAL/Apollonius_graph_hierarchy_2.h>
#include <CGAL/Apollonius_graph_filtered_traits_2.h>


// Type definition for the traits class.
// In this example we explicitly define the exact kernel. We also
// explicitly define what operations to use for the evaluation of the
// predicates and constructions, when the filtering and the exact
// kernels are used respectively.
// Note that the operations allowed for the filtering and the
// constructions (field operations plus square roots) are different
// from the operations allowed when the exact kernel is used (ring
// operations).
typedef CGAL::Field_with_sqrt_tag  CM;
typedef CGAL::Integral_domain_without_division_tag        EM;
typedef CGAL::Apollonius_graph_filtered_traits_2<CK,CM,EK,EM> Traits;

// Now we use the Apollonius graph hierarchy.
// The hierarchy is faster for inputs consisting of about more than
// 1,000 sites
typedef CGAL::Apollonius_graph_hierarchy_2<Traits> Apollonius_graph;
```

```
int main()
{
  std::ifstream ifs("data/hierarchy.cin");
  assert( ifs );

  Apollonius_graph ag;
  Apollonius_graph::Site_2 site;

  // read the sites and insert them in the Apollonius graph
  while ( ifs >> site ) {
    ag.insert(site);
  }

  // validate the Apollonius graph
  assert( ag.is_valid(true, 1) );

  return 0;
}
```

File: examples/Apollonius_graph_2/ag2_hierarchy.cpp

2D Apollonius Graphs (Delaunay Graphs of Disks) Reference Manual

Menelaos Karavelas and Mariette Yvinec

An Apollonius graph is the dual of the Apollonius diagram, also known as the *additively weighted Voronoi diagram*. It is essentially the Voronoi diagram of a set of disks, where the distance of a point of the plane from a disk is defined as the Euclidean distance of the point and the center of the circle, minus the radius of the disk.

CGAL provides the class *CGAL::Apollonius_graph_2<Gt,Agds>* for computing the 2D Apollonius graph. The two template parameters must be models of the *ApolloniusGraphTraits_2* and *ApolloniusGraphDataStructure_2* concepts. The first concept is related to the geometric objects and predicates associated with Apollonius graphs, whereas the second concept refers to the data structure used to represent the Apollonius graph. The classes *Apollonius_graph_traits_2<K,Method_tag>* and *Triangulation_data_structure_2<Vb,Fb>* are models of the aforementioned concepts.

45.6 Classified Reference Pages

Concepts

Classes

45.7 Alphabetical List of Reference Pages

CGAL::Apollonius_graph_2<Gt,Agds>

Definition

The class *Apollonius_graph_2<Gt,Agds>* represents the Apollonius graph. It supports insertions and deletions of sites. It is templated by two template arguments *Gt*, which must be a model of *ApolloniusGraphTraits_2*, and *Agds*, which must be a model of *ApolloniusGraphDataStructure_2*. The second template argument defaults to *CGAL::Triangulation_data_structure_2< CGAL::Apollonius_graph_vertex_base_2<Gt,true>, CGAL::Triangulation_face_base_2<Gt> >*.

#include <CGAL/Apollonius_graph_2.h>

Is Model for the Concepts

DelaunayGraph_2

Types

typedef Agds	*Data_structure;*	A type for the underlying data structure.
typedef Data_structure	*Triangulation_data_structure;*	Same as the *Data_structure* type. This type has been introduced in order for the *Apollonius_graph_2<Gt,Agds>* class to be a model of the *DelaunayGraph_2* concept.
typedef Gt	*Geom_traits;*	A type for the geometric traits.
typedef Gt::Point_2	*Point_2;*	A type for the point defined in the geometric traits.
typedef Gt::Site_2	*Site_2;*	A type for the Apollonius site, defined in the geometric traits.

The vertices and faces of the Apollonius graph are accessed through *handles*, *iterators* and *circulators*. The iterators and circulators are all bidirectional and non-mutable. The circulators and iterators are assignable to the corresponding handle types, and they are also convertible to the corresponding handles. The edges of the Apollonius graph can also be visited through iterators and circulators, the edge circulators and iterators are also bidirectional and non-mutable. In the following, we call *infinite* any face or edge incident to the infinite vertex and the infinite vertex itself. Any other feature (face, edge or vertex) of the Apollonius graph is said to be *finite*. Some iterators (the *All* iterators) allow to visit finite or infinite features while the others (the *Finite* iterators) visit only finite features. Circulators visit both infinite and finite features.

typedef Data_structure::Edge	*Edge;*	the edge type. The *Edge(f,i)* is the edge common to faces *f* and *f.neighbor(i)*. It is also the edge joining the vertices *vertex(cw(i))* and *vertex(ccw(i))* of *f*. *Precondition*: *i* must be *0*, *1* or *2*.
typedef Data_structure::Vertex	*Vertex;*	A type for a vertex.
typedef Data_structure::Face	*Face;*	A type for a face.
typedef Data_structure::Vertex_handle	*Vertex_handle;*	A type for a handle to a vertex.
typedef Data_structure::Face_handle	*Face_handle;*	A type for a handle to a face.
typedef Data_structure::Vertex_circulator	*Vertex_circulator;*	A type for a circulator over vertices incident to a given vertex.

typedef Data_structure::Face_circulator	*Face_circulator;*	A type for a circulator over faces incident to a given vertex.
typedef Data_structure::Edge_circulator	*Edge_circulator;*	A type for a circulator over edges incident to a given vertex.
typedef Data_structure::Vertex_iterator	*All_vertices_iterator;*	A type for an iterator over all vertices.
typedef Data_structure::Face_iterator	*All_faces_iterator;*	A type for an iterator over all faces.
typedef Data_structure::Edge_iterator	*All_edges_iterator;*	A type for an iterator over all edges.
typedef Data_structure::size_type	*size_type;*	An unsigned integral type.

Apollonius_graph_2<Gt,Agds>:: Finite_vertices_iterator	A type for an iterator over finite vertices.
Apollonius_graph_2<Gt,Agds>:: Finite_faces_iterator	A type for an iterator over finite faces.
Apollonius_graph_2<Gt,Agds>:: Finite_edges_iterator	A type for an iterator over finite edges.

In addition to iterators and circulators for vertices and faces, iterators for sites are provided. In particular there are iterators for the entire set of sites, the hidden sites and the visible sites of the Apollonius graph.

Apollonius_graph_2<Gt,Agds>:: Sites_iterator	A type for an iterator over all sites.
Apollonius_graph_2<Gt,Agds>:: Visible_sites_iterator	A type for an iterator over all visible sites.
Apollonius_graph_2<Gt,Agds>:: Hidden_sites_iterator	A type for an iterator over all hidden sites.

Creation

Apollonius_graph_2<Gt,Agds> ag(Gt gt=Gt()); Creates an Apollonius graph using *gt* as geometric traits.

template< class Input_iterator >
Apollonius_graph_2<Gt,Agds> ag(Input_iterator first, Input_iterator beyond, Gt gt=Gt());

Creates an Apollonius graph using *gt* as geometric traits and inserts all sites in the range [*first*, *beyond*).
Precondition: *Input_iterator* must be a model of *InputIterator*. The value type of *Input_iterator* must be *Site_2*.

Apollonius_graph_2<Gt,Agds> ag(other); Copy constructor. All faces and vertices are duplicated. After the construction, *ag* and *other* refer to two different Apollonius graphs : if *other* is modified, *ag* is not.

Apollonius_graph_2<Gt,Agds> ag = other Assignment. If *ag* and *other* are the same object nothing is done. Otherwise, all the vertices and faces are duplicated. After the assignment, *ag* and *other* refer to different Apollonius graphs : if *other* is modified, *ag* is not.

Access Functions

Geom_traits	*ag.geom_traits()*	Returns a reference to the Apollonius graph traits object.
Data_structure	*ag.data_structure()*	Returns a reference to the underlying data structure.
Data_structure	*ag.tds()*	Same as *data_structure()*. This method has been added in compliance with the *DelaunayGraph_2* concept.
int	*ag.dimension()*	Returns the dimension of the Apollonius graph.
size_type	*ag.number_of_vertices()*	Returns the number of finite vertices.
size_type	*ag.number_of_visible_sites()*	Returns the number of visible sites.

size_type	*ag.number_of_hidden_sites()*	Returns the number of hidden sites.
size_type	*ag.number_of_faces()*	Returns the number of faces (both finite and infinite) of the Apollonius graph.
Face_handle	*ag.infinite_face()*	Returns a face incident to the *infinite_vertex*.
Vertex_handle	*ag.infinite_vertex()*	Returns the *infinite_vertex*.
Vertex_handle	*ag.finite_vertex()*	Returns a vertex distinct from the *infinite_vertex*. *Precondition*: The number of (visible) vertices in the Apollonius graph must be at least one.

Traversal of the Apollonius graph

An Apollonius graph can be seen as a container of faces and vertices. Therefore the Apollonius graph provides several iterators and circulators that allow to traverse it (completely or partially).

Face, Edge and Vertex Iterators

The following iterators allow respectively to visit finite faces, finite edges and finite vertices of the Apollonius graph. These iterators are non-mutable, bidirectional and their value types are respectively *Face*, *Edge* and *Vertex*. They are all invalidated by any change in the Apollonius graph.

Finite_vertices_iterator	*ag.finite_vertices_begin()*	Starts at an arbitrary finite vertex.
Finite_vertices_iterator	*ag.finite_vertices_end()*	Past-the-end iterator.
Finite_edges_iterator	*ag.finite_edges_begin()*	Starts at an arbitrary finite edge.
Finite_edges_iterator	*ag.finite_edges_end()*	Past-the-end iterator.
Finite_faces_iterator	*ag.finite_faces_begin()*	Starts at an arbitrary finite face.
Finite_faces_iterator	*ag.finite_faces_end() const*	Past-the-end iterator.

The following iterators allow respectively to visit all (both finite and infinite) faces, edges and vertices of the Apollonius graph. These iterators are non-mutable, bidirectional and their value types are respectively *Face*, *Edge* and *Vertex*. They are all invalidated by any change in the Apollonius graph.

All_vertices_iterator	*ag.all_vertices_begin()*	Starts at an arbitrary vertex.
All_vertices_iterator	*ag.all_vertices_end()*	Past-the-end iterator.
All_edges_iterator	*ag.all_edges_begin()*	Starts at an arbitrary edge.
All_edges_iterator	*ag.all_edges_end()*	Past-the-end iterator.
All_faces_iterator	*ag.all_faces_begin()*	Starts at an arbitrary face.
All_faces_iterator	*ag.all_faces_end()*	Past-the-end iterator.

Site iterators

The following iterators allow respectively to visit all sites, the visible sites and the hidden sites. These iterators are non-mutable, bidirectional and their value type is *Site_2*. They are all invalidated by any change in the Apollonius graph.

Sites_iterator	*ag.sites_begin()*	Starts at an arbitrary site.

Sites_iterator	*ag.sites_end()*	Past-the-end iterator.
Visible_sites_iterator	*ag.visible_sites_begin()*	Starts at an arbitrary visible site.
Visible_sites_iterator	*ag.visible_sites_end()*	Past-the-end iterator.
Hidden_sites_iterator	*ag.hidden_sites_begin()*	Starts at an arbitrary hidden site.
Hidden_sites_iterator	*ag.hidden_sites_end()*	Past-the-end iterator.

Face, Edge and Vertex Circulators

The Apollonius graph also provides circulators that allow to visit respectively all faces or edges incident to a given vertex or all vertices adjacent to a given vertex. These circulators are non-mutable and bidirectional. The operator *operator++* moves the circulator counterclockwise around the vertex while the *operator--* moves clockwise. A face circulator is invalidated by any modification of the face pointed to. An edge circulator is invalidated by any modification in one of the two faces incident to the edge pointed to. A vertex circulator is invalidated by any modification in any of the faces adjacent to the vertex pointed to.

Face_circulator *ag.incident_faces(Vertex_handle v)*

Starts at an arbitrary face incident to v.

Face_circulator *ag.incident_faces(Vertex_handle v, Face_handle f)*

Starts at face f.
Precondition: Face f is incident to vertex v.

Edge_circulator *ag.incident_edges(Vertex_handle v)*

Starts at an arbitrary edge incident to v.

Edge_circulator *ag.incident_edges(Vertex_handle v, Face_handle f)*

Starts at the first edge of f incident to v, in counterclockwise order around v.
Precondition: Face f is incident to vertex v.

Vertex_circulator *ag.incident_vertices(Vertex_handle v)*

Starts at an arbitrary vertex incident to v.

Vertex_circulator *ag.incident_vertices(Vertex_handle v, Face_handle f)*

Starts at the first vertex of f adjacent to v in counterclockwise order around v.
Precondition: Face f is incident to vertex v.

Traversal of the Convex Hull

Applied on the *infinite_vertex* the above functions allow to visit the vertices on the convex hull and the infinite edges and faces. Note that a counterclockwise traversal of the vertices adjacent to the *infinite_vertex* is a clockwise traversal of the convex hull.

Vertex_circulator *ag.incident_vertices(ag.infinite_vertex())*
Vertex_circulator *ag.incident_vertices(ag.infinite_vertex(), Face_handle f)*
Face_circulator *ag.incident_faces(ag.infinite_vertex())*

Face_circulator *ag.incident_faces(ag.infinite_vertex(), Face_handle f)*
Edge_circulator *ag.incident_edges(ag.infinite_vertex())*
Edge_circulator *ag.incident_edges(ag.infinite_vertex(), Face_handle f)*

Predicates

The class *Apollonius_graph_2<Gt,Agds>* provides methods to test the finite or infinite character of any feature.

bool *ag.is_infinite(Vertex_handle v) const*

true, iff *v* is the *infinite_vertex*.

bool *ag.is_infinite(Face_handle f) const* *true*, iff face *f* is infinite.

bool *ag.is_infinite(Face_handle f, int i) const*

true, iff edge *(f,i)* is infinite.

bool *ag.is_infinite(Edge e) const* *true*, iff edge *e* is infinite.

bool *ag.is_infinite(Edge_circulator ec) const*

true, iff edge **ec* is infinite.

Insertion

template< class Input_iterator >
unsigned int *ag.insert(Input_iterator first, Input_iterator beyond)*

Inserts the sites in the range [*first*,*beyond*). The number of sites in the range [*first*, *beyond*) is returned.
Precondition: *Input_iterator* must be a model of *InputIterator* and its value type must be *Site_2*.

Vertex_handle *ag.insert(Site_2 s)* Inserts the site *s* in the Apollonius graph. If *s* is visible then the vertex handle of *s* is returned, otherwise *Vertex_handle(NULL)* is returned.

Vertex_handle *ag.insert(Site_2 s, Vertex_handle vnear)*

Inserts *s* in the Apollonius graph using the site associated with *vnear* as an estimate for the nearest neighbor of the center of *s*. If *s* is visible then the vertex handle of *s* is returned, otherwise *Vertex_handle(NULL)* is returned.

Removal

void *ag.remove(Vertex_handle v)* Removes the site associated to the vertex handle *v* from the Apollonius graph.
Precondition: *v* must correspond to a valid finite vertex of the Apollonius graph.

Nearest neighbor location

Vertex_handle	*ag.nearest_neighbor(Point_2 p)*	Finds the nearest neighbor of the point p. In other words it finds the site whose Apollonius cell contains p. Ties are broken arbitrarily and one of the nearest neighbors of p is returned. If there are no visible sites in the Apollonius diagram *Vertex_handle(NULL)* is returned.
Vertex_handle	*ag.nearest_neighbor(Point_2 p, Vertex_handle vnear)*	Finds the nearest neighbor of the point p using the site associated with *vnear* as an estimate for the nearest neighbor of p. Ties are broken arbitrarily and one of the nearest neighbors of p is returned. If there are no visible sites in the Apollonius diagram *Vertex_handle(NULL)* is returned.

Access to the dual

The *Apollonius_graph_2* class provides access to the duals of the faces of the graph. The dual of a face of the Apollonius graph is a site. If the originating face is infinite, its dual is a site with center at infinity (or equivalently with infinite weight), which means that it can be represented geometrically as a line. If the originating face is finite, its dual is a site with finite center and weight. In the following three methods the returned object is assignable to either *Site_2* or *Gt::Line_2*, depending on whether the corresponding face of the Apollonius graph is finite or infinite, respectively.

Gt::Object_2	*ag.dual(Face_handle f) const*	Returns the dual corresponding to the face handle f. The returned object can be assignable to one of the following: *Site_2*, *Gt::Line_2*.
Gt::Object_2	*ag.dual(All_faces_iterator it) const*	Returns the dual of the face to which *it* points to. The returned object can be assignable to one of the following: *Site_2*, *Gt::Line_2*.
Gt::Object_2	*ag.dual(Finite_faces_iterator it) const*	Returns the dual of the face to which *it* points to. The returned object can be assignable to one of the following: *Site_2*, *Gt::Line_2*.

I/O

template< class Stream >

Stream&	*ag.draw_primal(Stream& str)*	Draws the Apollonius graph to the stream *str*. *Precondition*: The following operators must be defined: *Stream& operator<<(Stream&, Gt::Segment_2)*, *Stream& operator<<(Stream&, Gt::Ray_2)*.

template < class Stream >

Stream&	*ag.draw_dual(Stream& str)*	Draws the dual of the Apollonius graph, i.e., the Apollonius diagram, to the stream *str*. *Precondition*: The following operators must be defined: *Stream& operator<<(Stream&, Gt::Segment_2)*, *Stream& operator<<(Stream&, Gt::Ray_2)*, *Stream& operator<<(Stream&, Gt::Line_2)*.

template< class Stream >

Stream&	*ag.draw_primal_edge(Edge e, Stream& str)*	Draws the edge *e* of the Apollonius graph to the stream *str*. *Precondition*: The following operators must be defined: *Stream& operator<<(Stream&, Gt::Segment_2)*, *Stream& operator<<(Stream&, Gt::Ray_2)*.

template< class Stream >

Stream&	*ag.draw_dual_edge(Edge e, Stream& str)*	Draws the dual of the edge *e* to the stream *str*. The dual of *e* is an edge of the Apollonius diagram. *Precondition*: The following operators must be defined: *Stream& operator<<(Stream&, Gt::Segment_2)*, *Stream& operator<<(Stream&, Gt::Ray_2)*, *Stream& operator<<(Stream&, Gt::Line_2)*.
void	*ag.file_output(std::ostream& os)*	Writes the current state of the Apollonius graph to an output stream. In particular, all visible and hidden sites are written as well as the underlying combinatorial data structure.
void	*ag.file_input(std::istream& is)*	Reads the state of the Apollonius graph from an input stream.
std::ostream&	*std::ostream& os << ag*	Writes the current state of the Apollonius graph to an output stream.
std::istream&	*std::istream& is >> ag*	Reads the state of the Apollonius graph from an input stream.

Validity check

bool	*ag.is_valid(bool verbose = false, int level = 1)*	Checks the validity of the Apollonius graph. If *verbose* is *true* a short message is sent to *std::cerr*. If *level* is 0, only the data structure is validated. If *level* is 1, then both the data structure and the Apollonius graph are validated. Negative values of *level* always return true, and values greater then 1 are equivalent to *level* being 1.

Miscellaneous

void	*ag.clear()*	Clears all contents of the Apollonius graph.
void	*ag.swap(other)*	The Apollonius graphs *other* and *ag* are swapped. *ag.swap(other)* should be preferred to *ag= other* or to *ag(other)* if *other* is deleted afterwards.

See Also

DelaunayGraph_2
ApolloniusGraphTraits_2
ApolloniusGraphDataStructure_2
ApolloniusGraphVertexBase_2
TriangulationFaceBase_2
CGAL::Apollonius_graph_traits_2<K,Method_tag>
CGAL::Apollonius_graph_filtered_traits_2<CK,CM,EK,EM,FK,FM>
CGAL::Triangulation_data_structure_2<Vb,Fb>
CGAL::Apollonius_graph_vertex_base_2<Gt,StoreHidden>
CGAL::Triangulation_face_base_2<Gt>

Concept

ApolloniusSite_2

Definition

The concept *ApolloniusSite_2* provides the requirements for an Apollonius site class.

Types

ApolloniusSite_2:: Point_2	The point type.
ApolloniusSite_2:: FT	The field number type.
ApolloniusSite_2:: RT	The ring number type.
ApolloniusSite_2:: Weight	The weight type. *Precondition*: It must be the same as *FT*.

Creation

ApolloniusSite_2 s(Point_2 p=Point_2(), Weight w= Weight(0));

ApolloniusSite_2 s(other);	Copy constructor.

Access Functions

Point_2	*s.point() const*	Returns the center of the Apollonius site.
Weight	*s.weight() const*	Returns the weight of the Apollonius site.

See Also

ApolloniusGraphTraits_2
CGAL::Apollonius_site_2<K>
CGAL::Apollonius_graph_traits_2<K,Method_tag>
CGAL::Apollonius_graph_filtered_traits_2<CK,CM,EK,EM,FK,FM>

CGAL::Apollonius_site_2<K>

Class

Definition

The class *Apollonius_site_2<K>* is a model for the concept *ApolloniusSite_2*. It is parametrized by a template parameter *K* which must be a model of the *Kernel* concept.

#include <CGAL/Apollonius_site_2.h>

Is Model for the Concepts

ApolloniusSite_2

Types

The class *Apollonius_site_2<K>* does not introduce any types in addition to the concept *ApolloniusSite_2*.

Creation

Apollonius_site_2<K>	*s(Point_2 p=Point_2(), Weight w= Weight(0));*	
Apollonius_site_2<K>	*s(other);*	Copy constructor.

I/O

The I/O operators are defined for *iostream*.

std::ostream&	*std::ostream& os << s*	Inserts the Apollonius site *s* into the stream *os*. *Precondition*: The insert operator must be defined for *Point_2* and *Weight*.
std::istream&	*std::istream& is >> s*	Reads an Apollonius site from the stream *is* and assigns it to *s*. *Precondition*: The extract operator must be defined for *Point_2* and *Weight*.

The information output in the *iostream* is: the point of the Apollonius site and its weight.

#include <CGAL/IO/Qt_widget_Apollonius_site_2.h>

Qt_widget&	*Qt_widget& w << s*	Inserts the Apollonius site *s* into the *Qt_widget* stream *w*. *Precondition*: The insert operator must be defined for *K::Circle_2*.

See Also

Kernel
ApolloniusSite_2
CGAL::Qt_widget
CGAL::Apollonius_graph_traits_2<K,Method_tag>
CGAL::Apollonius_graph_filtered_traits_2<CK,CM,EK,EM,FK,FM>

ApolloniusGraphDataStructure_2

Definition

The concept *ApolloniusGraphDataStructure_2* refines the concept *TriangulationDataStructure_2*. In addition it provides two methods for the insertion and removal of a degree 2 vertex in the data structure. The insertion method adds a new vertex to the specified edge, thus creating two new edges. Moreover, it creates two new faces that have the two newly created edges in common (see figure below). The removal method performs the reverse operation.

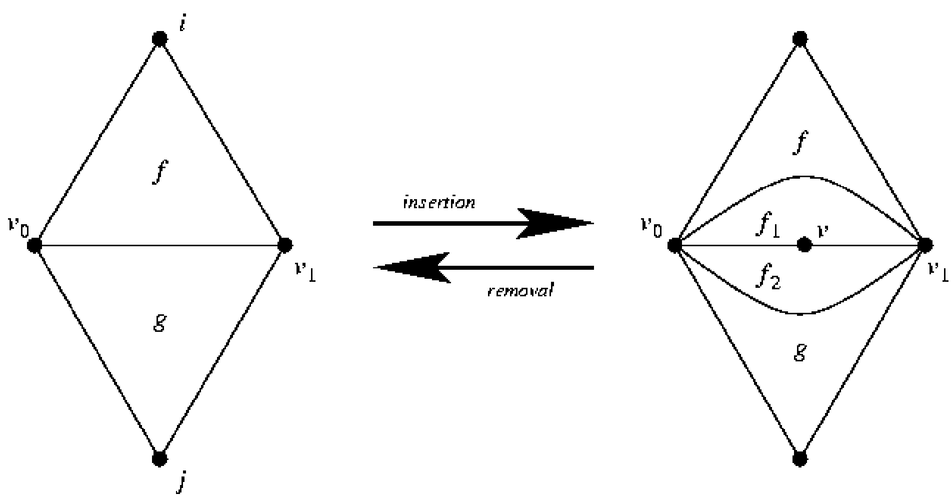


Figure 45.4: Insertion and removal of degree 2 vertices. Left to right: The edge *(f,i)* is replaced by two edges by means of inserting a vertex v on the edge. The faces f_1 and f_2 are created. Right to left: the faces f_1 and f_2 are destroyed. The vertex v is deleted and its two adjacent edges are merged.

We only describe the additional requirements with respect to the *TriangulationDataStructure_2* concept.

Refines

TriangulationDataStructure_2

Insertion

Vertex_handle	*agds.insert_degree_2(Face_handle f, int i)*	Inserts a degree two vertex and two faces adjacent to it that have two common edges. The edge defined by the face handle f and the integer i is duplicated. It returns a handle to the vertex created.

Removal

void	*agds.remove_degree_2(Vertex_handle v)*	Removes a degree 2 vertex and the two faces adjacent to it. The two edges of the star of *v* that are not incident to it are collapsed. *Precondition*: The degree of *v* must be equal to 2.

Has Models

CGAL::Triangulation_data_structure_2<Vb,Fb>

See Also

TriangulationDataStructure_2
ApolloniusGraphVertexBase_2
TriangulationFaceBase_2

Concept

ApolloniusGraphVertexBase_2

Definition

The concept *ApolloniusGraphVertexBase_2* describes the requirements for the vertex base class of the *ApolloniusGraphDataStructure_2* concept. A vertex stores an Apollonius site and provides access to one of its incident faces through a *Face_handle*. In addition, it maintains a container of sites. The container stores the hidden sites related to the vertex.

Types

ApolloniusGraphVertexBase_2:: Geom_traits	A type for the geometric traits that defines the site stored. *Precondition*: The type *Geom_traits* must define the type *Site_2*.
ApolloniusGraphVertexBase_2:: Store_hidden	A Boolean that indicates if hidden sites are actually stored or not. Its value is *true* if hidden sites are stored, *false* otherwise.
ApolloniusGraphVertexBase_2:: Site_2	A type for the site stored. *Precondition*: This type must coincide with the type *Geom_traits::Site_2*.
ApolloniusGraphVertexBase_2:: Apollonius_graph_data_structure_2	A type for the Apollonius graph data structure, to which the vertex belongs to.
ApolloniusGraphVertexBase_2:: Vertex_handle	A type for the vertex handle of the Apollonius graph data structure.
ApolloniusGraphVertexBase_2:: Face_handle	A type for the face handle of the Apollonius graph data structure.
ApolloniusGraphVertexBase_2:: Hidden_sites_iterator	An iterator that iterates over the hidden sites in the hidden sites container of the vertex. *Precondition*: Must be a model of *Iterator*.

Creation

ApolloniusGraphVertexBase_2 v;	Default constructor.
ApolloniusGraphVertexBase_2 v(Site_2 s);	Constructs a vertex associated with the Apollonius site *s* and embedded at the center of *s*.
ApolloniusGraphVertexBase_2 v(Site_2 s, Face_handle f);	Constructs a vertex associated with the site *s*, embedded at the center of *s*, and pointing to the face associated with the face handle *f*.

Access Functions

Site_2	*v.site()*	Returns the Apollonius site.
Face_handle	*v.face()*	Returns a handle to an incident face.
unsigned int	*v.number_of_hidden_sites()*	Returns the number of hidden sites in the hidden sites container.

Hidden_sites_iterator	*v.hidden_sites_begin()*	Starts at an arbitrary hidden site.
Hidden_sites_iterator	*v.hidden_sites_end()*	Past-the-end iterator.

Setting and unsetting

void	*v.set_site(Site_2 s)*	Sets the Apollonius site.
void	*v.set_face(Face_handle f)*	Sets the incident face.
void	*v.add_hidden_site(Site_2 s)*	Adds a hidden site to the container of hidden sites.
void	*v.clear_hidden_sites_container()*	Clears the container of hidden sites.

Checking

bool

v.is_valid(bool verbose, int level) const

Performs any required tests on a vertex.

Has Models

CGAL::Apollonius_graph_vertex_base_2<Gt,StoreHidden>.

See Also

ApolloniusGraphDataStructure_2
ApolloniusGraphTraits_2
CGAL::Apollonius_graph_vertex_base_2<Gt,StoreHidden>

Class

CGAL::Apollonius_graph_vertex_base_2<Gt,StoreHidden>

Definition

The class *Apollonius_graph_vertex_base_2<Gt,StoreHidden>* provides a model for the *ApolloniusGraphVertexBase_2* concept which is the vertex base required by the *ApolloniusGraphDataStructure_2* concept. The class *Apollonius_graph_vertex_base_2<Gt,StoreHidden>* has two template arguments, the first being the geometric traits of the Apollonius graph and should be a model of the concept *ApolloniusGraphTraits_2*. The second is a Boolean which controls whether hidden sites are actually stored. Such a control is important if the user is not interested in hidden sites and/or if only insertions are made, in which case no hidden site can become visible. If *StoreHidden* is set to *true*, hidden sites are stored, otherwise they are discarded. By default *StoreHidden* is set to *true*.

#include <CGAL/Apollonius_graph_vertex_base_2.h>

Is Model for the Concepts

ApolloniusGraphVertexBase_2

Creation

Apollonius_graph_vertex_base_2<Gt,StoreHidden> vb;	Default constructor.
Apollonius_graph_vertex_base_2<Gt,StoreHidden> vb(Site_2 s);	Constructs a vertex associated with the site *s* and embedded at the center of *s*.
Apollonius_graph_vertex_base_2<Gt,StoreHidden> vb(Site_2 s, Face_handle f);	Constructs a vertex associated with the site *s*, embedded at the center of *s*, and pointing to the face associated with the face handle *f*.

See Also

ApolloniusGraphVertexBase_2
ApolloniusGraphDataStructure_2
ApolloniusGraphTraits_2
CGAL::Triangulation_data_structure_2<Vb,Fb>
CGAL::Apollonius_graph_traits_2<K,Method_tag>
CGAL::Apollonius_graph_filtered_traits_2<CK,CM,EK,EM,FK,FM>

ApolloniusGraphTraits_2

Definition

The concept *ApolloniusGraphTraits_2* provides the traits requirements for the *Apollonius_graph_2* class. In particular, it provides a type *Site_2*, which must be a model of the concept *ApolloniusSite_2*. It also provides constructions for sites and several function object types for the predicates.

Types

ApolloniusGraphTraits_2:: Point_2	A type for a point.
ApolloniusGraphTraits_2:: Site_2	A type for an Apollonius site. Must be a model of the concept *ApolloniusSite_2*.
ApolloniusGraphTraits_2:: Line_2	A type for a line. Only required if access to the dual of the Apollonius graph is required or if the primal or dual diagram are inserted in a stream.
ApolloniusGraphTraits_2:: Ray_2	A type for a ray. Only required if access to the dual of the Apollonius graph is required or if the primal or dual diagram are inserted in a stream.
ApolloniusGraphTraits_2:: Segment_2	A type for a segment. Only required if access to the dual of the Apollonius graph is required or if the primal or dual diagram are inserted in a stream.
ApolloniusGraphTraits_2:: Object_2	A type representing different types of objects in two dimensions, namely: *Point_2*, *Site_2*, *Line_2*, *Ray_2* and *Segment_2*.
ApolloniusGraphTraits_2:: FT	A type for the field number type of sites.
ApolloniusGraphTraits_2:: RT	A type for the ring number type of sites.
ApolloniusGraphTraits_2:: Assign_2	Must provide *template <class T> bool operator() (T& t, Object_2 o)* which assigns *o* to *t* if *o* was constructed from an object of type *T*. Returns *true*, if the assignment was possible.

ApolloniusGraphTraits_2:: Construct_object_2

Must provide *template <class T> Object_2 operator()(T t)* that constructs an object of type *Object_2* that contains *t* and returns it.

ApolloniusGraphTraits_2:: Construct_Apollonius_vertex_2

A constructor for a point of the Apollonius diagram equidistant from three sites. Must provide *Point_2 operator()(Site_2 s1, Site_2 s2, Site_2 s3)*, which constructs a point equidistant from the sites *s1*, *s2* and *s3*.

ApolloniusGraphTraits_2:: Construct_Apollonius_site_2

A constructor for a dual Apollonius site (a site whose center is a vertex of the Apollonius diagram and its weight is the common distance of its center from the three defining sites). Must provide *Site_2 operator()(Site_2 s1, Site_2 s2, Site_2 s3)*, which constructs a dual site whose center *c* is equidistant from *s1*, *s2* and *s3*, and its weight is equal to the (signed) distance of *c* from *s1* (or *s2* or *s3*).
Must also provide *Line_2 operator()(Site_2 s1, Site_2 s2)*, which constructs a line bitangent to *s1* and *s2*. This line is the dual site of *s1*, *s2* and the site at infinity; it can be viewed as a dual Apollonius site whose center is at infinity and its weight is infinite.

ApolloniusGraphTraits_2:: Compare_x_2

A predicate object type. Must provide *Comparison_result operator()(Site_2 s1, Site_2 s2)*, which compares the x-coordinates of the centers of *s1* and *s2*.

ApolloniusGraphTraits_2:: Compare_y_2

A predicate object type. Must provide *Comparison_result operator()(Site_2 s1, Site_2 s2)*, which compares the y-coordinates of the centers of *s1* and *s2*.

ApolloniusGraphTraits_2:: Compare_weight_2

A predicate object type. Must provide *Comparison_result operator()(Site_2 s1, Site_2 s2)*, which compares the weights of *s1* and *s2*.

ApolloniusGraphTraits_2:: Orientation_2

A predicate object type. Must provide *Orientation operator()(Site_2 s1, Site_2 s2, Site_2 s3)*, which performs the usual orientation test for the centers of the three sites *s1*, *s2* and *s3*.
Must also provide *Orientation operator()(Site_2 s1, Site_2 s2, Site_2 s3, Site_2 p1, Site_2 p2)*, which performs the usual orientation test for the Apollonius vertex of *s1*, *s2*, *s3* and the centers of *p1* and *p2*.
Precondition: the Apollonius vertex of *s1*, *s2* and *s3* must exist.

ApolloniusGraphTraits_2:: Is_hidden_2

A predicate object type. Must provide *bool operator()(Site_2 s1, Site_2 s2)*, which returns *true* if the circle corresponding to *s2* is contained in the closure of the disk corresponding to *s1*, *false* otherwise.

ApolloniusGraphTraits_2:: Oriented_side_of_bisector_2

A predicate object type. Must provide *Oriented_side operator()(Site_2 s1, Site_2 s2, Point_2 p)*, which returns the oriented side of the bisector of *s1* and *s2* that contains *p*. Returns *ON_POSITIVE_SIDE* if *p* lies in the half-space of *s1* (i.e., *p* is closer to *s1* than *s2*); returns *ON_NEGATIVE_SIDE* if *p* lies in the half-space of *s2*; returns *ON_ORIENTED_BOUNDARY* if *p* lies on the bisector of *s1* and *s2*.

ApolloniusGraphTraits_2:: Vertex_conflict_2

A predicate object type. Must provide *Sign operator()(Site_2 s1, Site_2 s2, Site_2 s3, Site_2 q)*, which returns the sign of the distance of *q* from the dual Apollonius site of *s1*, *s2*, *s3*.
Precondition: the dual Apollonius site of *s1*, *s2*, *s3* must exist.
Must also provide *Sign operator()(Site_2 s1, Site_2 s2, Site_2 q)*, which returns the sign of the distance of *q* from the bitangent line of *s1*, *s2* (a degenerate dual Apollonius site, with its center at infinity).

ApolloniusGraphTraits_2:: Finite_edge_interior_conflict_2

A predicate object type. Must provide *bool operator()(Site_2 s1, Site_2 s2, Site_2 s3, Site_2 s4, Site_2 q, bool b)*. The sites *s1*, *s2*, *s3* and *s4* define an Apollonius edge that lies on the bisector of *s1* and *s2* and has as endpoints the Apollonius vertices defined by the triplets *s1*, *s2*, *s3* and *s1*, *s4* and *s2*. The Boolean *b* denotes if the two Apollonius vertices are in conflict with the site *q* (in which case *b* should be *true*, otherwise *false*). In case that *b* is *true*, the predicate returns *true* if and only if the entire Apollonius edge is in conflict with *q*. If *b* is *false*, the predicate returns *false* if and only if *q* is not in conflict with the Apollonius edge.
Precondition: the Apollonius vertices of *s1*, *s2*, *s3*, and *s1*, *s4*, *s2* must exist.
Must also provide *bool operator()(Site_2 s1, Site_2 s2, Site_2 s3, Site_2 q, bool b)*. The sites *s1*, *s2*, *s3* and the site at infinity s_∞ define an Apollonius edge that lies on the bisector of *s1* and *s2* and has as endpoints the Apollonius vertices defined by the triplets *s1*, *s2*, *s3* and *s1*, s_∞ and *s2* (the second Apollonius vertex is actually at infinity). The Boolean *b* denotes if the two Apollonius vertices are in conflict with the site *q* (in which case *b* should be *true*, otherwise *false*). In case that *b* is *true*, the predicate returns *true* if and only if the entire Apollonius edge is in conflict with *q*. If *b* is *false*, the predicate returns *false* if and only if *q* is not in conflict with the Apollonius edge.
Precondition: the Apollonius vertex of *s1*, *s2*, *s3* must exist.
Must finally provide *bool operator()(Site_2 s1, Site_2 s2, Site_2 q, bool b)*. The sites *s1*, *s2* and the site at infinity s_∞ define an Apollonius edge that lies on the bisector of *s1* and *s2* and has as endpoints the Apollonius vertices defined by the triplets *s1*, *s2*, s_∞ and *s1*, s_∞ and *s2* (both Apollonius vertices are actually at infinity). The Boolean *b* denotes if the two Apollonius vertices are in conflict with the site *q* (in which case *b* should be *true*, otherwise *false*). In case that *b* is *true*, the predicate returns *true* if and only if the entire Apollonius edge is in conflict with *q*. If *b* is *false*, the predicate returns *false* if and only if *q* is not in conflict with the Apollonius edge.

ApolloniusGraphTraits_2:: Infinite_edge_interior_conflict_2

A predicate object type. Must provide *bool operator()(Site_2 s1, Site_2 s2, Site_2 s3, Site_2 q, bool b)*. The sites s_∞, *s1*, *s2* and *s3* define an Apollonius edge that lies on the bisector of s_∞ and *s1* and has as endpoints the Apollonius vertices defined by the triplets s_∞, *s1*, *s2* and s_∞, *s3* and *s1*. The Boolean *b* denotes if the two Apollonius vertices are in conflict with the site *q* (in which case *b* should be *true*, otherwise *false*. In case that *b* is *true*, the predicate returns *true* if and only if the entire Apollonius edge is in conflict with *q*. If *b* is *false*, the predicate returns *false* if and only if *q* is not in conflict with the Apollonius edge.

ApolloniusGraphTraits_2:: Is_degenerate_edge_2

A predicate object type. Must provide *bool operator()(Site_2 s1, Site_2 s2, Site_2 s3, Site_2 s4)*. It returns *true* if the Apollonius edge defined by *s1*, *s2*, *s3* and *s4* is degenerate, *false* otherwise. An Apollonius edge is called degenerate if its two endpoints coincide.
Precondition: the Apollonius vertices of *s1*, *s2*, *s3*, and *s1*, *s4*, *s2* must exist.

Creation

ApolloniusGraphTraits_2 gt;	Default constructor.
ApolloniusGraphTraits_2 gt(other);	Copy constructor.

ApolloniusGraphTraits_2	*gt = other*	Assignment operator.

Access to predicate objects

Compare_x_2	*gt.compare_x_2_object()*
Compare_y_2	*gt.compare_y_2_object()*
Compare_weight_2	*gt.compare_weight_2_object()*
Orientation_2	*gt.orientation_2_object()*
Is_hidden_2	*gt.is_hidden_2_object()*
Oriented_side_of_bisector_2	*gt.oriented_side_of_bisector_test_2_object()*
Vertex_conflict_2	*gt.vertex_conflict_2_object()*
Finite_edge_interior_conflict_2	*gt.finite_edge_interior_conflict_2_object()*
Infinite_edge_interior_conflict_2	*gt.infinite_edge_interior_conflict_2_object()*
Is_degenerate_edge_2	*gt.is_degenerate_edge_2_object()*

Access to constructor objects

Construct_object_2	*gt.construct_object_2_object()*
Construct_Apollonius_vertex_2	*gt.construct_Apollonius_vertex_2_object()*
Construct_Apollonius_site_2	*gt.construct_Apollonius_site_2_object()*

Access to other objects

Assign_2	*gt.assign_2_object()*

Has Models

CGAL::Apollonius_graph_traits_2<K,Method_tag>
CGAL::Apollonius_graph_filtered_traits_2<CK,CM,EK,EM,FK,FM>

See Also

CGAL::Apollonius_graph_2<Gt,Agds>
CGAL::Apollonius_graph_traits_2<K,Method_tag>
CGAL::Apollonius_graph_filtered_traits_2<CK,CM,EK,EM,FK,FM>

CGAL::Apollonius_graph_traits_2<K,Method_tag>

Definition

The class *Apollonius_graph_traits_2<K,Method_tag>* provides a model for the *ApolloniusGraphTraits_2* concept. This class has two template parameters. The first template parameter must be a model of the *Kernel* concept. The second template parameter corresponds to how predicates are evaluated. There are two predefined possible values for *Method_tag*, namely *CGAL::Sqrt_field_tag* and *CGAL::Ring_tag*. The first one must be used when the number type used in the representation supports the exact evaluation of signs of expressions involving all four basic operations and square roots, whereas the second one requires the exact evaluation of signs of ring-type expressions, i.e., expressions involving only additions, subtractions and multiplications. The default value for *Method_tag* is *CGAL::Ring_tag*. The way the predicates are evaluated is discussed in [KE02, KE03].

#include <CGAL/Apollonius_graph_traits_2.h>

Is Model for the Concepts

ApolloniusGraphTraits_2

Creation

Apollonius_graph_traits_2<K,Method_tag>	*traits*;	Default constructor.
Apollonius_graph_traits_2<K,Method_tag>	*traits(other);*	Copy constructor.
Apollonius_graph_traits_2<K,Method_tag>	*traits = other*	Assignment operator.

See Also

Kernel
ApolloniusGraphTraits_2
CGAL::Ring_tag
CGAL::Sqrt_field_tag
CGAL::Apollonius_graph_2<Gt,Agds>
CGAL::Apollonius_graph_filtered_traits_2<CK,CM,EK,EM,FK,FM>

Class

CGAL::Apollonius_graph_filtered_traits_2<CK,CM,EK,EM,FK,FM>

Definition

The class *Apollonius_graph_filtered_traits_2<CK,CM,EK,EM,FK,FM>* provides a model for the *ApolloniusGraphTraits_2* concept.

The class *Apollonius_graph_filtered_traits_2<CK,CM,EK,EM,FK,FM>* uses the filtering technique [BBP01] to achieve traits for the *Apollonius_graph_2<Gt,Agds>* class with efficient and exact predicates given an exact kernel *EK* and a filtering kernel *FK*. The geometric constructions associated provided by this class are equivalent to those provided by the traits class *Apollonius_graph_traits_2<CK,CM>*, which means that they may be inexact.

This class has six template parameters. The first, third and fifth template parameters must be a models of the *Kernel* concept. The second, fourth and sixth template parameters correspond to how predicates are evaluated. There are two predefined possible values for *Method_tag*, namely *CGAL::Sqrt_field_tag* and *CGAL::Ring_tag*. The first one must be used when the number type used in the representation supports the exact evaluation of signs of expressions involving all four basic operations and square roots, whereas the second one requires the exact evaluation of signs of ring-type expressions, i.e., expressions involving only additions, subtractions and multiplications. The way the predicates are evaluated is discussed in [KE02, KE03].

The default values for the template parameters are as follows: *CM = CGAL::Ring_tag*, *EK = CGAL::Simple_cartesian<CGAL::MP_Float>*, *EM = CM*, *FK = CGAL::Simple_cartesian<CGAL::Interval_nt<false> >*, *FM = CM*.

#include <CGAL/Apollonius_graph_filtered_traits_2.h>

Is Model for the Concepts

ApolloniusGraphTraits_2

Creation

Apollonius_graph_filtered_traits_2<CK,CM,EK,EM,FK,FM>	*traits*;	Default constructor.
Apollonius_graph_filtered_traits_2<CK,CM,EK,EM,FK,FM>	*traits(other);*	Copy constructor.
Apollonius_graph_filtered_traits_2<CK,CM,EK,EM,FK,FM>	*traits = other*	Assignment operator.

See Also

Kernel
ApolloniusGraphTraits_2
CGAL::Ring_tag
CGAL::Sqrt_field_tag
CGAL::Apollonius_graph_2<Gt,Agds>
CGAL::Apollonius_graph_traits_2<K,Method_tag>

CGAL::Apollonius_graph_hierarchy_2<Gt,Agds>

Definition

We provide an alternative to the class *Apollonius_graph_2<Gt,Agds>* for the dynamic construction of the Apollonius graph. The *Apollonius_graph_hierarchy_2<Gt,Agds>* class maintains a hierarchy of Apollonius graphs. The bottom-most level of the hierarchy contains the full Apollonius diagram. A site that is in level i, is in level $i+1$ with probability $1/\alpha$ where $\alpha > 1$ is some constant. The difference between the *Apollonius_graph_2<Gt,Agds>* class and the *Apollonius_graph_hierarchy_2<Gt,Agds>* is on how the nearest neighbor location is done. Given a point p the location is done as follows: at the top most level we find the nearest neighbor of p as in the *Apollonius_graph_2<Gt,Agds>* class. At every subsequent level i we use the nearest neighbor found at level $i+1$ to find the nearest neighbor at level i. This is a variant of the corresponding hierarchy for points found in [Dev98]. The class has two template parameters which have essentially the same meaning as in the *Apollonius_graph_2<Gt,Agds>* class. The first template parameter must be a model of the *ApolloniusGraphTraits_2* concept. The second template parameter must be a model of the *ApolloniusGraphDataStructure_2* concept. However, the vertex base class that is to be used in the Apollonius graph data structure must be a model of the *ApolloniusGraphHierarchyVertexBase_2* concept. The second template parameter defaults to *Triangulation_data_structure_2< Apollonius_graph_hierarchy_vertex_base_2< Apollonius_graph_vertex_base_2<Gt,true> >, Triangulation_face_base_2<Gt> >*.

The *Apollonius_graph_hierarchy_2<Gt,Agds>* class derives publicly from the *Apollonius_graph_2<Gt,Agds>* class. The interface is the same with its base class. In the sequel only the methods overridden are documented.

#include <CGAL/Apollonius_graph_hierarchy_2.h>

Inherits From

CGAL::Apollonius_graph_2<Gt,Agds>

Types

Apollonius_graph_hierarchy_2<Gt,Agds> does not introduce other types than those introduced by its base class *Apollonius_graph_2<Gt,Agds>*.

Creation

Apollonius_graph_hierarchy_2<Gt,Agds> agh(Gt gt=Gt());

Creates an hierarchy of Apollonius graphs using *gt* as geometric traits.

template< class Input_iterator >
Apollonius_graph_hierarchy_2<Gt,Agds> agh(Input_iterator first, Input_iterator beyond, Gt gt=Gt());

Creates an Apollonius graph hierarchy using *gt* as geometric traits and inserts all sites in the range [*first*, *beyond*).

Apollonius_graph_hierarchy_2<Gt,Agds> agh(other);

Copy constructor. All faces, vertices and inter-level pointers are duplicated. After the construction, *agh* and *other* refer to two different Apollonius graph hierarchies: if *other* is modified, *agh* is not.

Apollonius_graph_hierarchy_2<Gt,Agds> *agh = other*

Assignment. All faces, vertices and inter-level pointers are duplicated. After the construction, *agh* and *other* refer to two different Apollonius graph hierarchies: if *other* is modified, *agh* is not.

Insertion

template< class Input_iterator >
unsigned int *agh.insert(Input_iterator first, Input_iterator beyond)*

Inserts the sites in the range [*first*,*beyond*). The number of sites in the range [*first*, *beyond*) is returned.
Precondition: *Input_iterator* must be a model of *InputIterator* and its value type must be *Site_2*.

Vertex_handle *agh.insert(Site_2 s)*

Inserts the site *s* in the Apollonius graph hierarchy. If *s* is visible then the vertex handle of *s* is returned, otherwise *Vertex_handle(NULL)* is returned.

Vertex_handle *agh.insert(Site_2 s, Vertex_handle vnear)*

Inserts *s* in the Apollonius graph hierarchy using the site associated with *vnear* as an estimate for the nearest neighbor of the center of *s*. If *s* is visible then the vertex handle of *s* is returned, otherwise *Vertex_handle(NULL)* is returned. A call to this method is equivalent to *agh.insert(s);* and it has been added for the sake of conformity with the interface of the *Apollonius_graph_2<Gt,Agds>* class.

Removal

void *agh.remove(Vertex_handle v)*

Removes the site associated to the vertex handle *v* from the Apollonius graph hierarchy.
Precondition: *v* must correspond to a valid finite vertex of the Apollonius graph hierarchy.

Nearest neighbor location

Vertex_handle *agh.nearest_neighbor(Point p)*

Finds the nearest neighbor of the point *p*. In other words it finds the site whose Apollonius cell contains *p*. Ties are broken arbitrarily and one of the nearest neighbors of *p* is returned. If there are no visible sites in the Apollonius diagram *Vertex_handle(NULL)* is returned.

Vertex_handle *agh.nearest_neighbor(Point p, Vertex_handle vnear)*

Finds the nearest neighbor of the point *p*. If there are no visible sites in the Apollonius diagram *Vertex_handle(NULL)* is returned. A call to this method is equivalent to *agh.nearest_neighbor(p);* and it has been added for the sake of conformity with the interface of the *Apollonius_graph_2<Gt,Agds>* class.

I/O

void	*agh.file_output(std::ostream& os)*	Writes the current state of the Apollonius graph hierarchy to an output stream. In particular, all visible and hidden sites are written as well as the underlying combinatorial hierarchical data structure.
void	*agh.file_input(std::istream& is)*	Reads the state of the Apollonius graph hierarchy from an input stream.
std::ostream&	*std::ostream& os << agh*	Writes the current state of the Apollonius graph hierarchy to an output stream.
std::istream&	*std::istream& is >> agh*	Reads the state of the Apollonius graph hierarchy from an input stream.

Validity check

bool *agh.is_valid(bool verbose = false, int level = 1) const*

Checks the validity of the Apollonius graph hierarchy. If *verbose* is *true* a short message is sent to *std::cerr*. If *level* is 0, the data structure at all levels is validated, as well as the inter-level pointers. If *level* is 1, then the data structure at all levels is validated, the inter-level pointers are validated and all levels of the Apollonius graph hierarchy are also validated. Negative values of *level* always return *true*, and values greater then 1 are equivalent to *level* being 1.

Miscellaneous

void	*agh.clear()*	Clears all contents of the Apollonius graph hierarchy.
void	*agh.swap(other)*	The Apollonius graph hierarchies *other* and *agh* are swapped. *agh.swap(other)* should be preferred to *agh*= *other* or to *agh(other)* if *other* is deleted afterwards.

See Also

ApolloniusGraphDataStructure_2
ApolloniusGraphTraits_2
ApolloniusGraphHierarchyVertexBase_2
CGAL::Apollonius_graph_2<Gt,Agds>
CGAL::Triangulation_data_structure_2<Vb,Fb>
CGAL::Apollonius_graph_traits_2<K,Method_tag>
CGAL::Apollonius_graph_filtered_traits_2<CK,CM,EK,EM,FK,FM>
CGAL::Apollonius_graph_hierarchy_vertex_base_2<Agvb>

Concept

ApolloniusGraphHierarchyVertexBase_2

Definition

The vertex of an Apollonius graph included in an Apollonius graph hierarchy has to provide some pointers to the corresponding vertices in the graphs of the next and preceding levels. Therefore, the concept *ApolloniusGraphHierarchyVertexBase_2* refines the concept *ApolloniusGraphVertexBase_2*, by adding two vertex handles to the corresponding vertices for the next and previous level graphs.

Refines

ApolloniusGraphVertexBase_2

Types

ApolloniusGraphHierarchyVertexBase_2 does not introduce any types in addition to those of *ApolloniusGraphVertexBase_2*.

Creation

ApolloniusGraphHierarchyVertexBase_2 v;	Default constructor.
ApolloniusGraphHierarchyVertexBase_2 v(Site_2 s);	Constructs a vertex associated with the site s and embedded at the center of s.
ApolloniusGraphHierarchyVertexBase_2 v(Site_2 s, Face_handle f);	Constructs a vertex associated with the site s, embedded at the center of s, and pointing to face f.

Operations

Vertex_handle	*v.up()*	Returns a handle to the corresponding vertex of the next level Apollonius graph. If such a vertex does not exist *Vertex_handle(NULL)* is returned.
Vertex_handle	*v.down()*	Returns a handle to the corresponding vertex of the previous level Apollonius graph.
void	*v.set_up(Vertex_handle u)*	Sets the handle for the vertex of the next level Apollonius graph.
void	*v.set_down(Vertex_handle d)*	Sets the handle for the vertex of the previous level Apollonius graph;

Has Models

CGAL::Apollonius_graph_hierarchy_vertex_base_2<CGAL::Apollonius_graph_vertex_base_2< Gt,StoreHidden> >

See Also

ApolloniusGraphDataStructure_2
ApolloniusGraphVertexBase_2
CGAL::Apollonius_graph_hierarchy_2<Gt,Agds>
CGAL::Triangulation_data_structure_2<Vb,Fb>
CGAL::Apollonius_graph_vertex_base_2<Gt,StoreHidden>
CGAL::Apollonius_graph_hierarchy_vertex_base_2<Agvb>

Class

CGAL::Apollonius_graph_hierarchy_vertex_base_2<Agvb>

Definition

The class *Apollonius_graph_hierarchy_vertex_base_2<Agvb>* provides a model for the *ApolloniusGraphHierarchyVertexBase_2* concept, which is the vertex base required by the *Apollonius_graph_hierarchy_2<Gt,Agds>* class. The class *Apollonius_graph_hierarchy_vertex_base_2<Agvb>* is templated by a class *Agvb* which must be a model of the *ApolloniusGraphVertexBase_2* concept.

#include <CGAL/Apollonius_graph_hierarchy_vertex_base_2.h>

Inherits From

Agvb

Is Model for the Concepts

ApolloniusGraphHierarchyVertexBase_2

Creation

Apollonius_graph_hierarchy_vertex_base_2<Agvb> hvb;	Default constructor.
Apollonius_graph_hierarchy_vertex_base_2<Agvb> hvb(Site_2 s);	Constructs a vertex associated with the site s and embedded at the center of s.
Apollonius_graph_hierarchy_vertex_base_2<Agvb> hvb(Site_2 s, Face_handle f);	Constructs a vertex associated with the site s, embedded at the center of s, and pointing to the face associated with the face handle f.

See Also

ApolloniusGraphVertexBase_2
ApolloniusGraphHierarchyVertexBase_2
CGAL::Apollonius_graph_vertex_base_2<Gt,StoreHidden>

Chapter 46

2D Voronoi Diagram Adaptor

Menelaos Karavelas

Contents

This chapter describes an adaptor that adapts two-dimensional triangulated Delaunay graphs to the corresponding Voronoi diagrams. We start with a few definitions and a description of the issues that this adaptor addresses in Section 46.1. The software design of the Voronoi diagram adaptor package is described in Section 46.2. In Section 46.3 we discuss the traits required for performing the adaptation, and finally in Section 46.5 we present a few examples using this adaptor.

46.1 Introduction

A Voronoi diagram is typically defined for a set of objects, also called sites in the sequel, that lie in some space Σ and a distance function that measures the distance of a point x in Σ from an object in the object set. In this package we are interested in planar Voronoi diagrams, so in the sequel the space Σ will be the space $\mathbb{R}^2$. Let $\mathcal{S} = \{S_1, S_2, \ldots, S_n\}$ be our set of sites and let $\delta(x, S_i)$ denote the distance of a point $x \in \mathbb{R}^2$ from the site S_i. Given two sites S_i and S_j, the set V_{ij} of points that are closer to S_i than to S_j with respect to the distance function $\delta(x, \cdot)$ is simply the set:

$$V_{ij} = \{x \in \mathbb{R}^2 : \delta(x, S_i) < \delta(x, S_j)\}.$$

We can then define the set V_i of points on the plane that are closer to S_i than to any other object in $\mathcal{S}$ as:

$$V_i = \bigcap_{i \neq j} V_{ij}.$$

The set V_i is said to be the *Voronoi cell* or *Voronoi face* of the site S_i. The locus of points on the plane that are equidistant from exactly two sites S_i and S_j is called a *Voronoi bisector*. A point that is equidistant to three or more objects in $\mathcal{S}$ is called a *Voronoi vertex*. A simply connected subset of a Voronoi bisector is called a *Voronoi edge*. The collection of Voronoi faces, edges and vertices is called the *Voronoi diagram* of the set $\mathcal{S}$ with respect to the distance function $\delta(x,\cdot)$, and it turns out that it is a subdivision of the plane, i.e., it is a planar graph.

We typically think of faces as 2-dimensional objects, edges as 1-dimensional objects and vertices as 0-dimensional objects. However, this may not be the case for several combinations of sites and distance functions (for example points in $\mathbb{R}^2$ under the L_1 or the L_∞ distance can produce 2-dimensional Voronoi edges). We call a Voronoi diagram *nice* if no such artifacts exist, i.e., if all vertices edges and faces are 0-, 1- and 2-dimensional, respectively.

Even nice Voronoi diagrams can end up being not so nice. The cell of a site can in general consist of several disconnected components. Such a case can happen, for example, when we consider weighted points $Q_i = (p_i, \lambda_i)$, where $p_i \in \mathbb{R}^2$, $\lambda_i \in \mathbb{R}$, and the distance function is the Euclidean distance multiplied by the weight of each site, i.e., $\delta_M(x, Q_i) = \lambda_i \|x - p_i\|$, where $\|\cdot\|$ denotes the Euclidean norm. In this package we are going to restrict ourselves to nice Voronoi diagrams that have the property that the Voronoi cell of each site is a simply connected region of the plane. We are going to call such Voronoi diagrams *simple Voronoi diagrams*. Examples of simple Voronoi diagrams include the usual Euclidean Voronoi diagram of points, the Euclidean Voronoi diagram of a set of disks on the plane (i.e., the Apollonius diagram), the Euclidean Voronoi diagram of a set of disjoint convex objects on the plane, or the power or (Laguerre) diagram for a set of circles on the plane. In fact every instance of an *abstract Voronoi diagram* in the sense of Klein [Kle89] is a simple Voronoi diagram in our setting. In the sequel when we refer to Voronoi diagrams we will refer to simple Voronoi diagrams.

In many cases we are not really interested in computing the Voronoi diagram itself, but rather its dual graph, called the *Delaunay graph*. In general the Delaunay graph is a planar graph, each face of which consists of at least three edges. Under the non-degeneracy assumption that no point on the plane is equidistant, under the distance function, to more than three sites, the Delaunay graph is a planar graph with triangular faces. In certain cases this graph can actually be embedded with straight line segments in which case we talk about a triangulation. This is the case, for example, for the Euclidean Voronoi diagram of points, or the power diagram of a set of circles. The dual graphs are, respectively, the Delaunay triangulation and the regular triangulation of the corresponding site sets. Graphs of non-constant non-uniform face complexity can be undesirable in many applications, so typically we end up triangulating the non-triangular faces of the Delaunay graph. Intuitively this amounts to imposing an implicit or explicit perturbation scheme during the construction of the Delaunay graph, that perturbs the input sites in such a way so as not to have degenerate configurations.

Choosing between computing the Voronoi diagram or the (triangulated) Delaunay graph is a major decision while implementing an algorithm. It heavily affects the design and choice of the different data structures involved. Although in theory the two approaches are entirely equivalent, it is not so straightforward to go from one representation to the other. The objective of this package is to provide a generic way of going from triangulated Delaunay graphs to planar subdivisions represented through a DCEL data structure. The goal is to provide an adaptor that gives the look and feel of a DCEL data structure, although internally it keeps a graph data structure representing triangular graphs.

The adaptation might seem straightforward at a first glance, and more or less this is case; after all one graph is the dual of the other. The situation becomes complicated whenever we want to treat artifacts of the representation used. Suppose for example that we have a set of sites that contains subsets of sites in degenerate positions. The computed triangulated Delaunay graph has triangular faces that may be the result of an implicit or explicit perturbation scheme. The dual of such a triangulated Delaunay graph is a Voronoi diagram that has all its vertices of degree 3, and for that purpose we are going to call it a *degree-3 Voronoi diagram* in order to distinguish it from the true Voronoi diagram of the input sites. A degree-3 Voronoi diagram can have degenerate features, namely Voronoi edges of zero length, and/or Voronoi faces of zero area. Although we can potentially treat such artifacts, they are nonetheless artifacts of the algorithm we used and do not correspond to the true geometry of the Voronoi diagram.

The manner that we treat such issues in this package in a generic way is by defining an *adaptation policy*. The adaptation policy is responsible for determining which features in the degree-3 Voronoi diagram are to be rejected and which not. The policy to be used can vary depending on the application or the intended usage of the resulting Voronoi diagram. What we care about is that firstly the policy itself is consistent and, secondly, that the adaptation is also done in a consistent manner. The latter is the responsibility of the adaptor provided by this package, whereas the former is the responsibility of the implementor of a policy.

In this package we currently provide two types of adaptation policies. The first one is the simplest: we reject no feature of the degree-3 Voronoi diagram; we call such a policy an *identity policy* since the Voronoi diagram produced is identical to the degree-3 Voronoi diagram. The second type of policy eliminates the degenerate features from the degree-3 Voronoi diagram yielding the true geometry of the Voronoi diagram of the input sites; we call such policies *degeneracy removal policies*.

Delaunay graphs can be mutable or non-mutable. By mutable we mean that sites can be inserted or removed at any time, in an entirely on-line fashion. By non-mutable we mean that once the Delaunay graph has been created, no changes, with respect to the set of sites defining it, are allowed. If the Delaunay graph is a non-mutable one, then the Voronoi diagram adaptor is a non-mutable adaptor as well.

If the Delaunay graph is mutable then the question of whether the Voronoi diagram adaptor is also mutable is slightly more complex to answer. As long as the adaptation policy used does not maintain a state, the Voronoi diagram adaptor is a mutable one; this is the case, for example, with our identity policy or the degeneracy removal policies. If, however, the adaptor maintains a state, then whether it is mutable or non-mutable really depends on whether its state can be updated after every change in the Delaunay graph. Such policies are our caching degeneracy removal policies: some of them result in mutable adaptors others result in non-mutable ones. In Section 46.4 we discuss the issue in more detail.

46.2 Software Design

The *Voronoi_diagram_2<DG,AT,AP>* class is parameterized by three template parameters. The first one must be a model of the *DelaunayGraph_2* concept. It corresponds to the API required by an object representing a Delaunay graph. All classes of CGAL that represent Delaunay diagrams are models of this concept, namely, Delaunay triangulations, regular triangulations, Apollonius graphs and segment Delaunay graphs. The second template parameter must be a model of the *AdaptationTraits_2* concept. We discuss this concept in detail in Section 46.3. The third template parameter must be model of the *AdaptationPolicy_2* concept, which we discuss in detail in Section 46.4.

The *Voronoi_diagram_2<DG,AT,AP>* class has been intentionally designed to provide an API similar to the arrangements class in CGAL: Voronoi diagrams are special cases of arrangements after all. The API of the two classes, however, could not be identical. The reason is that arrangements in CGAL do not yet support more than one unbounded faces, or equivalently, cannot handle unbounded curves. On the contrary, a Voronoi diagram defined over at least two generating sites, has at least two unbounded faces.

On a more technical level, the *Voronoi_diagram_2<DG,AT,AP>* class imitates the representation of the Voronoi diagram (seen as a planar subdivision) by a DCEL (Doubly Connected Edge List) data structure. We have vertices (the Voronoi vertices), halfedges (oriented versions of the Voronoi edges) and faces (the Voronoi cells). In particular, we can basically perform every operation we can perform in a standard DCEL data structure:

- go from a halfedge to its next and previous in the face;
- go from one face to an adjacent one through a halfedge and its twin (opposite) halfedge;
- walk around the boundary of a face;
- enumerate/traverse the halfedges incident to a vertex
- from a halfedge, access the adjacent face;

- from a face, access an adjacent halfedges;
- from a halfedges, access its source and target vertices;
- from a vertex, access an incident halfedge.

In addition to the above possibilities for traversal, we can also traverse the following features through iterators:

- the vertices of the Voronoi diagram;
- the edges or halfedges of the Voronoi diagram;
- the faces of the Voronoi diagram;
- the bounded faces of the Voronoi diagram;
- the bounded halfedges of the Voronoi diagram;
- the unbounded faces of the Voronoi diagram;
- the unbounded halfedges of the Voronoi diagram;
- the sites defining the Voronoi diagram.

Finally, depending on the adaptation traits passed to the Voronoi diagram adaptor, we can perform point location queries, namely given a point p we can determine the feature of the Voronoi diagram (vertex, edge, face) on which p lies.

46.3 The Adaptation Traits

The *AdaptationTraits_2* concept defines the types and functors required by the adaptor in order to access geometric information in the Delaunay graph that is needed by the *Voronoi_diagram_2<DG,AT,AP>* class. In particular, it provides functors for accessing sites in the Delaunay graph and constructing Voronoi vertices from their dual faces in the Delaunay graph. Finally, it defines a tag that indicates whether nearest site queries are to be supported by the Voronoi diagram adaptor. If such queries are to be supported, a functor is required.

Given a query point, the nearest site functor should return information related to how many and which sites of the Voronoi diagram are at equal and minimal distance from the query point. In particular, if the query point is closest to a single site, the vertex handle of the Delaunay graph corresponding to this site is returned. If the query point is closest to exactly two site, the edge of the Delaunay graph that is dual to the Voronoi edges on which the query point lies is returned. If three (or more) sites are closest to the query point, then the query point coincides with a vertex in the Voronoi diagram, and the face handle of the face in the Delaunay graph that is dual to the Voronoi vertex is returned. This way of abstracting the point location mechanism allows for multiple different point location strategies, which are passed to the Voronoi diagram adaptor through different models of the *AdaptationTraits_2* concept. The point location and nearest sites queries of the *Voronoi_diagram_2< DG,AT,AP>* class use internally this nearest site query functor.

In this package we provide four adaptation traits classes, all of which support nearest site queries:

- The *Apollonius_graph_adaptation_traits_2<AG2>* class: it provides the adaptation traits for Apollonius graphs.
- The *Delaunay_triangulation_adaptation_traits_2<DT2>* class: it provides the adaptation traits for Delaunay triangulations.

- The *Regular_triangulation_adaptation_traits_2<RT2>* class: it provides the adaptation traits for regular triangulations.
- The *Segment_Delaunay_graph_adaptation_traits_2<SDG2>* class: it provides the adaptation traits for segment Delaunay graphs.

46.4 The Adaptation Policy

As mentioned above, when we perform the adaptation of a triangulated Delaunay graph to a Voronoi diagram, a question that arises is whether we want to eliminate certain features of the Delaunay graph when we construct its Voronoi diagram representation (such features could be the Voronoi edges of zero length or, for the Voronoi diagram of a set of segments forming a polygon, all edges outside the polygon). The manner that we treat such issues in this package in a generic way is by defining an adaptation policy. The adaptation policy is responsible for determining which features in the degree-3 Voronoi diagram are to be rejected and which not. The policy to be used can vary depending on the application or the intended usage of the resulting Voronoi diagram.

The concept *AdaptationPolicy_2* defines the requirements on the predicate functors that determine whether a feature of the triangulated Delaunay graph should be rejected or not. More specifically it defines an *Edge_rejector* and a *Face_rejector* functor that answer the question: "should this edge (face) of the Voronoi diagram be rejected?". In addition to the edge and face rejectors the adaptation policy defines a tag, the *Has_inserter* tag. This tag is either set to *CGAL::Tag_true* or to *CGAL::Tag_false*. Semantically it determines if the adaptor is allowed to insert sites in an on-line fashion (on-line removals are not yet supported). In the former case, i.e., when on-line site insertions are allowed, an additional functor is required, the *Site_inserter* functor. This functor takes a reference to a Delaunay graph and a site, and inserts the site in the Delaunay graph. Upon successful insertion, a handle to the vertex representing the site in the Delaunay graph is returned.

We have implemented two types of policies that provide two different ways for answering the question of which features of the Voronoi diagram to keep and which to discard. The first one is called the *identity policy* and corresponds to the *Identity_policy_2<DG,VT>* class. This policy is in some sense the simplest possible one, since it does not reject any feature of the Delaunay graph. The Voronoi diagram provided by the adaptor is the true dual (from the graph-theoretical point of view) of the triangulated Delaunay graph adapted. This policy assumes that the Delaunay graph adapted allows for on-line insertions, and the *Has_inserter* tag is set to *CGAL::Tag_true*. A default site inserter functor is also provided.

The second type of policy we provide is called *degeneracy removal policy*. If the set of sites defining the triangulated Delaunay graph contains subsets of sites in degenerate configurations, the graph-theoretical dual of the triangulated Delaunay graph has edges and potentially faces that are geometrically degenerate. By that we mean that the dual of the triangulated Delaunay graph can have Voronoi edges of zero length or Voronoi faces/cells of zero area. Such features may not be desirable and ideally we would like to eliminate them. The degeneracy removal policies eliminate exactly these features and provide a Voronoi diagram where all edges have non-zero length and all cells have non-zero area. More specifically, in these policies the *Edge_rejector* and *Face_rejector* functors reject the edges and vertices of the Delaunay graph that correspond to dual edges and faces that have zero length and area, respectively. In this package we provide four degeneracy removal policies, namely:

- The *Apollonius_graph_degeneracy_removal_policy_2<AG2>* class: it provides an adaptation policy for removing degeneracies when adapting an Apollonius graph to an Apollonius diagram.
- The *Delaunay_triangulation_degeneracy_removal_policy_2<DT2>* class: it provides an adaptation policy for removing degeneracies when adapting a Delaunay triangulation to a point Voronoi diagram.
- The *Regular_triangulation_degeneracy_removal_policy_2<RT2>* class: it provides an adaptation policy for removing degeneracies when adapting a regular triangulation to a power diagram

- The *Segment_Delaunay_graph_degeneracy_removal_policy_2<SDG2>* class: it provides an adaptation policy for removing degeneracies when adapting a segment Delaunay graph to a segment Voronoi diagram.

A variation of the degeneracy removal policies are the *caching degeneracy removal policies*. In these policies we cache the results of the edge and face rejectors. In particular, every time we want to determine, for example, if an edge of the Delaunay graph has, as dual edge in the Voronoi diagram, an edge of zero length, we check if the result has already been computed. If yes, we simply return the outcome. If not, we perform the necessary geometric tests, compute the answer, cache it and return it. Such a policy really pays off when we have a lot of degenerate data in our input set of sites. Verifying whether a Voronoi edge is degenerate or not implies computing the outcome of a predicate in a possibly degenerate or near degenerate configuration, which is typically very costly (compared to computing the same predicate in a generic configuration). To avoid this cost every single time we want to check if a Voronoi edge is degenerate or not, we compute the result of the geometric predicate the first time the adaptor asks for it, and simply lookup the answer in the future. In this package we provide four caching degeneracy removal policies, one per degeneracy removal policy mentioned above. Intentionally, we have not indicated the value of the *Has_inserter* tag for the degeneracy removal and caching degeneracy removal policies. The issue is discussed in detail in the sequel.

We raised the question above, as to whether the adaptor is a mutable or non-mutable one, in the sense of whether we can add/remove sites in an on-line fashion. The answer to this question depends on: (1) whether the Delaunay graph adapted allows for on-line insertions/removals and (2) whether the adaptation policies maintains a state and whether this state is easily maintainable when we want to allow for on-line modifications.

The way we indicate if we allow on-line insertions of sites is via the *Has_inserter* tag (as mentioned, on-line removals are currently not supported). The *Has_inserter* tag has two possible values, namely, *CGAL::Tag_true* and *CGAL::Tag_false*. The value *CGAL::Tag_true* indicates that the Delaunay graph allows for on-line insertions, whereas the value *CGAL::Tag_false* indicates the opposite. Note that these values *do not* indicate if the Delaunay graph supports on-line insertions, but rather whether the Voronoi diagram adaptor should be able to perform on-line insertions or not. This delicate point will be become clearer below.

Let us consider the various scenarios. If the Delaunay graph is non-mutable, the Voronoi diagram adaptor cannot perform on-line insertions of sites. In this case not only degeneracy removal policies, but rather every single adaptation policy for adapting the Delaunay graph in question should have the *Has_inserter* tag set to *CGAL::Tag_false*.

If the Delaunay graph is mutable, i.e., on-line site insertions as are allowed, we can choose between two types of adaptation policies, those that allow these on-line insertions and those that do not. In the former case the *Has_inserter* tag should be set to *CGAL::Tag_true*, whereas in the latter to *CGAL::Tag_false*. In other words, even if the Delaunay graph is mutable, we can choose (by properly determining the value of the *Has_inserter* tag) if the adaptor should be mutable as well. At a first glance it may seem excessive to restrict existing functionality. There are situations, however, where such a choice is necessary.

Consider a caching degeneracy removal policy. If we do not allow for on-line insertions then the cached quantities are always valid since the Voronoi diagram never changes. If we allow for on-line insertions the Voronoi diagram can change, which implies that the results of the edge and faces degeneracy testers that we have cached are no longer valid or relevant. In these cases, we need to somehow update these cached results, and ideally we would like to do this in an efficient manner. The inherent dilemma in the above discussion is whether the Voronoi diagram adaptor should be able to perform on-line insertions of sites. The answer to this question in this framework is given by the *Has_inserter* tag. If the tag is set to *CGAL::Tag_false* the adaptor cannot insert sites on-line, whereas if the tag is set to *CGAL::Tag_true* the adaptor can add sites on-line. In other words, the *Has_inserter* tag determines how the Voronoi diagram adaptor should behave, and this is enough from the adaptor's point of view.

From the point of a view of a policy writer the dilemma is still there: should the policy allow for on-line insertions or not? The answer really depends on what are the consequences of such a choice. For a policy that has no state, such as our degeneracy removal policies, it is natural to set the *Has_inserter* tag to *CGAL::Tag_true*. For our caching degeneracy removal policies, our choice was made on the grounds of whether we can update the cached results efficiently when insertions are performed. For CGAL's Apollonius graphs, Delaunay triangulation and regular triangulations it is possible to ask what are the edges and faces of the Delaunay graph that are to be destroyed when a query site is inserted. This is done via the *get_conflicts* method provided by these classes. Using the outcome of the *get_conflicts* method the site inserter can first update the cached results (i.e., indicate which are invalidated) and then perform the actual insertion. Such a method does not yet exist for segment Delaunay graphs. We have thus chosen to support on-line insertions for all non-caching degeneracy removal policies. The caching degeneracy removal policy for segment Delaunay graphs does not support on-line insertions, whereas the remaining three caching degeneracy removal policies support on-line insertions.

Efficiency Considerations

One last item that merits some discussion are the different choices from the point of view of time- and space-efficiency.

As far as the Voronoi diagram adaptor is concerned, only a copy of the adaptation traits and a copy of the adaptation policy are stored in it. The various adaptation traits classes we provide are empty classes (i.e., they do not store anything). The major time and space efficiency issues arise from the various implementations of the adaptation policies. Clearly, the identity policy has no dominant effect on neither the time or space efficiency. The costs when choosing this policy are due to the underlying Delaunay graph.

The non-caching degeneracy removal policies create a significant time overhead since every time we want to access a feature of the Voronoi diagram, we need to perform geometric tests in order to see if this feature or one of its neighboring ones has been rejected. Such a policy is acceptable if we know we are away from degeneracies or for small input sizes. In the case of the segment Delaunay graph, it is also the only policy we provide that at the same time removes degeneracies and allows for on-line insertion of sites. Caching policies seem to be the best choice for moderate to large input sizes (1000 sites and more). They do not suffer from the problem of dealing with degenerate configurations, but since they cache the results, they increase the space requirements by linear additive factor. To conclude, if the user is interested in getting a Voronoi diagram without degenerate features and knows all sites in advance, the best course of action is to insert all sites at construction time and use a caching degeneracy removal policy. This strategy avoids the updates of the cached results after each individual insertion, due to the features of the Voronoi diagram destroyed because of the site inserted.

46.5 Examples

In this section we present an example that shows how to perform point location queries.

```
#include <iostream>
#include <fstream>
#include <cassert>

// includes for defining the Voronoi diagram adaptor
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/Delaunay_triangulation_2.h>
#include <CGAL/Voronoi_diagram_2.h>
#include <CGAL/Delaunay_triangulation_adaptation_traits_2.h>
#include <CGAL/Delaunay_triangulation_adaptation_policies_2.h>

// typedefs for defining the adaptor
```

```
typedef CGAL::Exact_predicates_inexact_constructions_kernel
        K;
typedef CGAL::Delaunay_triangulation_2<K>
        DT;
typedef CGAL::Delaunay_triangulation_adaptation_traits_2<DT>
        AT;
typedef
    CGAL::Delaunay_triangulation_caching_degeneracy_removal_policy_2<DT>
    AP;
typedef CGAL::Voronoi_diagram_2<DT,AT,AP>
        VD;

// typedef for the result type of the point location
typedef AT::Site_2                    Site_2;
typedef AT::Point_2                   Point_2;

typedef VD::Locate_result             Locate_result;
typedef VD::Vertex_handle             Vertex_handle;
typedef VD::Face_handle               Face_handle;
typedef VD::Halfedge_handle           Halfedge_handle;
typedef VD::Ccb_halfedge_circulator   Ccb_halfedge_circulator;

void print_endpoint(Halfedge_handle e, bool is_src) {
  std::cout << "\t";
  if ( is_src ) {
    if ( e->has_source() )  std::cout << e->source()->point() <<
   std::endl;
    else  std::cout << "point at infinity" << std::endl;
  } else {
    if ( e->has_target() )  std::cout << e->target()->point() <<
   std::endl;
    else  std::cout << "point at infinity" << std::endl;
  }
}

int main()
{
  std::ifstream ifs("data/data1.dt.cin");
  assert( ifs );

  VD vd;

  Site_2 t;
  while ( ifs >> t ) { vd.insert(t); }
  ifs.close();

  assert( vd.is_valid() );

  std::ifstream ifq("data/queries1.dt.cin");
  assert( ifq );

  Point_2 p;
  while ( ifq >> p ) {
    std::cout << "Query point (" << p.x() << "," << p.y()
```

```
            << ") lies on a Voronoi " << std::flush;

    Locate_result lr = vd.locate(p);
    if ( Vertex_handle* v = boost::get<Vertex_handle>(&lr) ) {
      std::cout << "vertex." << std::endl;
      std::cout << "The Voronoi vertex is:" << std::endl;
      std::cout << "\t" << (*v)->point() << std::endl;
    } else if ( Halfedge_handle* e = boost::get<Halfedge_handle>(&lr) ) {
      std::cout << "edge." << std::endl;
      std::cout << "The source and target vertices "
                << "of the Voronoi edge are:" << std::endl;
      print_endpoint(*e, true);
      print_endpoint(*e, false);
    } else if ( Face_handle* f = boost::get<Face_handle>(&lr) ) {
      std::cout << "face." << std::endl;
      std::cout << "The vertices of the Voronoi face are"
                << " (in counterclockwise order):" << std::endl;
      Ccb_halfedge_circulator ec_start = (*f)->outer_ccb();
      Ccb_halfedge_circulator ec = ec_start;
      do {
        print_endpoint(ec, false);
      } while ( ++ec != ec_start );
    }
    std::cout << std::endl;
  }
  ifq.close();

  return 0;
}
```

File: examples/Voronoi_diagram_2/vd_2_point_location.cpp

2D Voronoi Diagram Adaptor Reference Manual

Menelaos Karavelas

CGAL provides the class *CGAL::Voronoi_diagram_2<DG,AT,AP>* for adapting the various (triangulated) Delaunay graphs to Voronoi diagrams according to some adaptation policy. In particular, the class *CGAL::Voronoi_diagram_2<DG,AT,AP>* provides an API for the duals of (triangulated) Delaunay graphs, that makes them look like planar subdivisions. The adaptation policy is responsible for deciding which edges and faces of these duals should be eliminated. This is especially important when, for instance, we want to eliminate degenerate features in the Voronoi diagram that are the result of the fact that Delaunay graphs are always triangulated and are due to degenerate configurations in the generating data.

The three template parameters must be models of the *DelaunayGraph_2*, *AdaptationTraits_2* and *AdaptationPolicy_2* concepts, respectively. The first concept is related to the Delaunay graphs that are to be adapted, whereas the second one is responsible for manipulating/accessing in a unified way the geometry of a specific Voronoi diagram as well as for performing nearest site queries. The third template parameter corresponds to the chosen adaptation policy and provides the necessary types and functors needed for performing this adaptation.

46.6 Classified Reference Pages

Concepts

Classes

46.7 Alphabetical List of Reference Pages

CGAL::Voronoi_diagram_2<DG,AT,AP>

Class

Definition

The class *Voronoi_diagram_2<DG,AT,AP>* provides an adaptor that enables us to view a triangulated Delaunay graph as their dual subdivision, the Voronoi diagram. The class *Voronoi_diagram_2<DG,AT,AP>* is designed to provide an API that is similar to that of CGAL's arrangements.
The first template parameter of the *Voronoi_diagram_2<DG,AT,AP>* class corresponds to the triangulated Delaunay graph and must be a model of the *DelaunayGraph_2* concept. The second template parameter must be a model of the *AdaptationTraits_2* concept. The third template parameter must be a model of the *AdaptationPolicy_2* concept. The third template parameter defaults to *CGAL::Identity_policy_2<DG,AT>*.

#include <CGAL/Voronoi_diagram_2.h>

Refines

DefaultConstructible, *CopyConstructible*, *Assignable*

Types

typedef DG	*Delaunay_graph;*	A type for the dual Delaunay graph.
typedef AT	*Adaptation_traits;*	A type for the adaptation traits needed by the Voronoi diagram adaptor.
typedef AP	*Adaptation_policy;*	A type for the adaptation policy used.
typedef Adaptation_traits::Point_2	*Point_2;*	A type a point.
typedef Adaptation_traits::Site_2	*Site_2;*	A type for the sites of the Voronoi diagram.
typedef Delaunay_graph::size_type	*size_type;*	A type for sizes.
typedef Delaunay_graph::Geom_traits	*Delaunay_geom_traits;*	A type for the geometric traits of the Delaunay graph.
typedef Delaunay_graph::Vertex_handle	*Delaunay_vertex_handle;*	A type for the vertex handles of the Delaunay graph.
typedef Delaunay_graph::Face_handle	*Delaunay_face_handle;*	A type for the face handles of the Delaunay graph.
typedef Delaunay_graph::Edge	*Delaunay_edge;*	A type for the edges of the Delaunay graph.

Voronoi_diagram_2<DG,AT,AP>:: Halfedge	A type for the halfedges of the Voronoi diagram.
Voronoi_diagram_2<DG,AT,AP>:: Vertex	A type for the vertices of the Voronoi diagram.
Voronoi_diagram_2<DG,AT,AP>:: Face	A type for the faces of the Voronoi diagram.

The vertices, edges and faces of the Voronoi diagram are accessed through *handles*, *iterators* and *circulators*. The iterators and circulators are all bidirectional and non-mutable. The circulators and iterators are assignable to the corresponding handle types, and they are also convertible to the corresponding handles.

Voronoi_diagram_2<DG,AT,AP>:: Halfedge_handle	Handle for halfedges.

Voronoi_diagram_2<DG,AT,AP>:: Vertex_handle	Handle for vertices.
Voronoi_diagram_2<DG,AT,AP>:: Face_handle	Handle for faces.
Voronoi_diagram_2<DG,AT,AP>:: Edge_iterator	A type for an iterator over Voronoi edges. Edges are considered non-oriented. Its value type is *Halfedge*.
Voronoi_diagram_2<DG,AT,AP>:: Halfedge_iterator	A type for an iterator over Voronoi halfedges. Halfedges are oriented and come in pairs. Its value type is *Halfedge*.
Voronoi_diagram_2<DG,AT,AP>:: Face_iterator	A type for an iterator over Voronoi faces. Its value type is *Face*.
Voronoi_diagram_2<DG,AT,AP>:: Vertex_iterator	A type for an iterator over Voronoi vertices. Its value type is *Vertex*.
Voronoi_diagram_2<DG,AT,AP>:: Halfedge_around_vertex_circulator	A type for a circulator over the halfedges that have a common vertex as their target. Its value type is *Halfedge*.
Voronoi_diagram_2<DG,AT,AP>:: Ccb_halfedge_circulator	A type for a circulator over the halfedges on the boundary of a Voronoi face. Its value type of is *Halfedge*.
Voronoi_diagram_2<DG,AT,AP>:: Unbounded_faces_iterator	A type for an iterator over the unbounded faces of the Voronoi diagram. Its value type is *Face*.
Voronoi_diagram_2<DG,AT,AP>:: Bounded_faces_iterator	A type for an iterator over the bounded faces of the Voronoi diagram. Its value type is *Face*.
Voronoi_diagram_2<DG,AT,AP>:: Unbounded_halfedges_iterator	A type for an iterator over the unbounded halfedges of the Voronoi diagram. Its value type is *Halfedge*.
Voronoi_diagram_2<DG,AT,AP>:: Bounded_halfedges_iterator	A type for an iterator over the bounded halfedges of the Voronoi diagram. Its value type is *Halfedge*.
Voronoi_diagram_2<DG,AT,AP>:: Site_iterator	A type for an iterator over the sites of the Voronoi diagram. Its value type is *Site_2*.

typedef boost::variant<Face_handle,Halfedge_handle,Vertex_handle>	*Locate_result;*	The result type of the point location queries.

Creation

Voronoi_diagram_2<DG,AT,AP> vd(Adaptation_traits at = Adaptation_traits(),
Adaptation_policy ap = Adaptation_policy(),
Delaunay_geom_traits gt = Delaunay_geom_traits())

Creates a Voronoi diagram using *at* as adaptation traits and *ap* as adaptation policy; the underlying Delaunay graph is created using *gt* as geometric traits.

Voronoi_diagram_2<DG,AT,AP> vd(Delaunay_graph dg,
bool swap_dg = false,
Adaptation_traits at = Adaptation_traits(),
Adaptation_policy ap = Adaptation_policy())

Creates a Voronoi diagram from the Delaunay graph *dg* and using *at* as adaptation traits and *ap* as adaptation policy. The Delaunay graph *dg* is fully copied if *swap_dg* is set to *false*, or swapped with the one stored internally if *swap_dg* is set to *true*.

template<class Iterator>
Voronoi_diagram_2<DG,AT,AP> vd(Iterator first,
Iterator beyond,
Adaptation_traits at = Adaptation_traits(),
Adaptation_policy ap = Adaptation_policy(),
Delaunay_geom_traits gt = Delaunay_geom_traits())

Creates a Voronoi diagram using as sites the sites in the iterator range *[first, beyond)*, *at* as adaptation traits and *ap* as adaptation policy; the underlying Delaunay graph is created using *gt* as geometric traits. *Iterator* must be a model of the *InputIterator* concept and its value type must be *Site_2*.

Access Methods

Delaunay_graph	*vd.dual()*	Returns a const reference to the dual graph, i.e., the Delaunay graph.
Halfedge_handle	*vd.dual(Delaunay_edge e)*	Returns a handle to the halfedge in the Voronoi diagram that is dual to the edge *e* in the Delaunay graph.
Face_handle	*vd.dual(Delaunay_vertex_handle v)*	Returns a handle to the face in the Voronoi diagram that is dual to the vertex corresponding to the vertex handle *v* in the Delaunay graph.
Vertex_handle	*vd.dual(Delaunay_face_handle f)*	Returns a handle to the vertex in the Voronoi diagram that is dual to the face corresponding to the face handle *f* in the Delaunay graph.
Adaptation_traits	*vd.adaptation_traits()*	Returns a reference to the Voronoi traits.
Adaptation_policy	*vd.adaptation_policy()*	Returns a reference to the adaptation policy.
size_type	*vd.number_of_vertices()*	Returns the number of Voronoi vertices.
size_type	*vd.number_of_faces()*	Returns the number of Voronoi faces (bounded and unbounded).
size_type	*vd.number_of_halfedges()*	Returns the number of halfedges (bounded and unbounded) in the Voronoi diagram. This is always an even number.
size_type	*vd.number_of_connected_components()*	Returns the number of connected components of the Voronoi skeleton.
Face_handle	*vd.unbounded_face()*	Returns one of the unbounded faces of the Voronoi diagram. If no unbounded faces exist (this can happen if the number of sites is zero) the default constructed face handle is returned.

Face_handle	*vd.bounded_face()*	Returns one of the bounded faces of the Voronoi diagram. If no bounded faces exist the default constructed face handle is returned.
Halfedge_handle	*vd.unbounded_halfedge()*	Returns one of the unbounded halfedges of the Voronoi diagram. If no unbounded halfedges exist the default constructed halfedge handle is returned.
Halfedge_handle	*vd.bounded_halfedge()*	Returns one of the bounded halfedges of the Voronoi diagram. If no bounded halfedges exist the default constructed halfedge handle is returned.

Traversal of the Voronoi diagram

A Voronoi diagram can be seen as a container of faces, vertices and halfedges. Therefore the Voronoi diagram provides several iterators and circulators that allow to traverse it.

Iterators

The following iterators allow respectively to visit the faces (all or only the unbounded/bounded ones), edges, halfedges (all or only the unbounded/bounded ones) and vertices of the Voronoi diagram. These iterators are non-mutable, bidirectional and their value types are respectively *Face*, *Halfedge*, *Halfedge* and *Vertex*. All iterators are convertible to the corresponding handles and are invalidated by any change in the Voronoi diagram.

Face_iterator	*vd.faces_begin()*	Starts at an arbitrary Voronoi face.
Face_iterator	*vd.faces_end()*	Past-the-end iterator.
Unbounded_faces_iterator	*vd.unbounded_faces_begin()*	Starts at an arbitrary unbounded Voronoi face.
Unbounded_faces_iterator	*vd.unbounded_faces_end()*	Past-the-end iterator.
Bounded_faces_iterator	*vd.bounded_faces_begin()*	Starts at an arbitrary bounded Voronoi face.
Bounded_faces_iterator	*vd.bounded_faces_end()*	Past-the-end iterator.
Edge_iterator	*vd.edges_begin()*	Starts at an arbitrary Voronoi edge.
Edge_iterator	*vd.edges_end()*	Past-the-end iterator.
Halfedge_iterator	*vd.halfedges_begin()*	Starts at an arbitrary Voronoi halfedge.
Halfedge_iterator	*vd.halfedges_end()*	Past-the-end iterator.
Unbounded_halfedges_iterator	*vd.unbounded_halfedges_begin()*	Starts at an arbitrary unbounded Voronoi edge.
Unbounded_halfedges_iterator	*vd.unbounded_halfedges_end()*	Past-the-end iterator.
Bounded_halfedges_iterator	*vd.bounded_halfedges_begin()*	Starts at an arbitrary bounded Voronoi edge.
Bounded_halfedges_iterator	*vd.bounded_halfedges_end()*	Past-the-end iterator.

Vertex_iterator	*vd.vertices_begin()*	Starts at an arbitrary Voronoi vertex.
Vertex_iterator	*vd.vertices_end()*	Past-the-end iterator.

The following iterator provides access to the sites that define the Voronoi diagram. Its value type is *Site_2*. It is invalidated by any change in the Voronoi diagram.

Site_iterator	*vd.sites_begin()*	Starts at an arbitrary site.
Site_iterator	*vd.sites_end()*	Past-the-end iterator.

Circulators

The Voronoi diagram adaptor also provides circulators that allow to visit all halfedges whose target is a given vertex – this is the *Halfedge_around_vertex_circulator*, as well as all halfedges on the boundary of a Voronoi face – this is the *Ccb_halfedge_circulator*. These circulators are non-mutable and bidirectional. The operator *operator++* moves the former circulator counterclockwise around the vertex while the *operator--* moves clockwise. The latter circulator is moved by the operator *operator++* to the next halfedge on the boundary in the counterclockwise sense, while *operator--* moves clockwise. When the *Ccb_halfedge_circulator* is defined over an infinite Voronoi face f, then applying *operator++* to a circulator corresponding to a halfedge whose target is not finite moves to the next infinite (or semi-infinite) halfedge of f in the counterclockwise sense. Similarly, applying *operator++* to a circulator corresponding to a halfedge whose source is not finite, moves to the previous infinite (or semi-infinite) halfedge of f in the clockwise sense. The *Halfedge_around_vertex_circulator* circulator is invalidated by any modification in the faces adjacent to the vertex over which it is defined. The *Ccb_halfedge_circulator* is invalidated by any modification in the face over which it is defined.

Ccb_halfedge_circulator	*vd.ccb_halfedges(Face_handle f)*	Returns a circulator over the halfedges on the boundary of f. The circulator is initialized to an arbitrary halfedge on the boundary of the Voronoi face f.
Ccb_halfedge_circulator	*vd.ccb_halfedges(Face_handle f, Halfedge_handle h)*	Returns a circulator over the halfedges on the boundary of f. The circulator is initialized with the halfedge h. *Precondition*: The halfedge h must lie on the boundary of f.
Halfedge_around_vertex_circulator	*vd.incident_halfedges(Vertex_handle v)*	Returns a circulator over the halfedges whose target is the Voronoi vertex v. The circulator is initialized to an arbitrary halfedge incident to v.
Halfedge_around_vertex_circulator	*vd.incident_halfedges(Vertex_handle v, Halfedge_handle h)*	Returns a circulator over the halfedges whose target is the Voronoi vertex v. The circulator is initialized with the halfedge h. *Precondition*: The vertex v must be the target vertex of the halfedge h.

Insertion

Face_handle	*vd.insert(Site_2 t)*	Inserts the site *t* in the Voronoi diagram. A handle to the face corresponding to the Voronoi face of *t* in the Voronoi diagram is returned. If *t* has an empty Voronoi cell, the default constructed face handle is returned. This method is supported only if *Voronoi_traits::Has_inserter* is set to *CGAL::Tag_true*.

template<class Iterator>
size_type *vd.insert(Iterator first, Iterator beyond)*

Inserts, in the Voronoi diagram, the sites in the iterator range *[first, beyond)*. The value type of *Iterator* must be *Site_2*. The number of sites in the iterator range is returned. This method is supported only if *Voronoi_traits::Has_inserter* is set to *CGAL::Tag_true*.

Queries

Locate_result	*vd.locate(Point_2 p)*	Performs point location for the query point *p*. In other words, the face, halfedge or vertex of the Voronoi diagram is found on which the point *p* lies. This method is supported only if *Voronoi_traits::Has_nearest_site_2* is set to *CGAL::Tag_true*. *Precondition*: The Voronoi diagram must contain at least one face.

I/O

void	*vd.file_output(std::ostream& os)*	Writes the current state of the Voronoi diagram to the output stream *os*. The following operator must be defined: *std::ostream& operator<<(std::ostream&, Delaunay_graph)*
void	*vd.file_input(std::istream& is)*	Reads the current state of the Voronoi diagram from the input stream *is*. The following operator must be defined: *std::istream& operator>>(std::istream&, Delaunay_graph)*
std::ostream&	*std::ostream& os << vd*	Writes the current state of the Voronoi diagram to the output stream *os*. The following operator must be defined: *std::ostream& operator<<(std::ostream&, Delaunay_graph)*
std::istream&	*std::istream& is >> vd*	Reads the current state of the Voronoi diagram from the input stream *is*. The following operator must be defined: *std::istream& operator>>(std::istream&, Delaunay_graph)*

Validity check

bool	*vd.is_valid()*	Checks the validity of the dual Delaunay graph and the Voronoi diagram adaptor.

Miscellaneous

void	*vd.clear()*	Clears all contents of the Voronoi diagram.
void	*vd.swap(other)*	The Voronoi diagrams *other* and *vd* are swapped. *vd.swap(other)* should be preferred to *vd= other* or to *vd(other)* if *other* is deleted afterwards.

See Also

DelaunayGraph_2
AdaptationTraits_2
AdaptationPolicy_2
CGAL::Voronoi_diagram_2<DG,AT,AP>::Face
CGAL::Voronoi_diagram_2<DG,AT,AP>::Halfedge
CGAL::Voronoi_diagram_2<DG,AT,AP>::Vertex
CGAL::Delaunay_triangulation_2<Traits,Tds>
CGAL::Regular_triangulation_2<Traits,Tds>
CGAL::Triangulation_hierarchy_2<Tr> provided that *Tr* is a model of *DelaunayGraph_2*
CGAL::Segment_Delaunay_graph_2<Gt,DS>
CGAL::Segment_Delaunay_graph_hierarchy_2<Gt,STag,DS>
CGAL::Apollonius_graph_2<Gt,Agds>
CGAL::Apollonius_graph_hierarchy_2<Gt,Agds>
CGAL::Apollonius_graph_adaptation_traits_2<AG2>
CGAL::Delaunay_triangulation_adaptation_traits_2<DT2>
CGAL::Regular_triangulation_adaptation_traits_2<RT2>
CGAL::Segment_Delaunay_graph_adaptation_traits_2<SDG2>
CGAL::Identity_policy_2<DG,AT>
CGAL::Apollonius_graph_degeneracy_removal_policy_2<AG2>
CGAL::Apollonius_graph_caching_degeneracy_removal_policy_2<AG2>
CGAL::Delaunay_triangulation_degeneracy_removal_policy_2<DT2>
CGAL::Delaunay_triangulation_caching_degeneracy_removal_policy_2<DT2>
CGAL::Regular_triangulation_degeneracy_removal_policy_2<RT2>
CGAL::Regular_triangulation_caching_degeneracy_removal_policy_2<RT2>
CGAL::Segment_Delaunay_graph_degeneracy_removal_policy_2<SDG2>
CGAL::Segment_Delaunay_graph_caching_degeneracy_removal_policy_2<SDG2>

Class

CGAL::Voronoi_diagram_2<DG,AT,AP>::Halfedge

Definition

The class *Halfedge* is the class provided by the *Voronoi_diagram_2<DG,AT,AP>* class for Voronoi halfedges. Below we present its interface.

Is Model for the Concepts

DefaultConstructible, *CopyConstructible*, *Assignable*, *EqualityComparable*, *LessThanComparable*

Types

Halfedge:: Vertex		A type for the vertices of the Voronoi diagram.
Halfedge:: Face		A type for the faces of the Voronoi diagram.
Halfedge:: Vertex_handle		Handle for the vertices of the Voronoi diagram.
Halfedge:: Face_handle		Handle for the faces of the Voronoi diagram.
Halfedge:: Halfedge_handle		Handle for the halfedges of the Voronoi diagram.
Halfedge:: Ccb_halfedge_circulator		A type for a bidirectional circulator over the halfedges of the boundary of a Voronoi face. The value type of the circulator is *CGAL::Voronoi_diagram_2< DG,AT,AP>::Halfedge* and is convertible to *Halfedge_handle*.
Halfedge:: Delaunay_graph		A type for the Delaunay graph. It is a model of the *DelaunayGraph_2* concept.
typedef Delaunay_graph::Edge	*Delaunay_edge;*	A type for the dual edge in the Delaunay graph.
typedef Delaunay_graph::Vertex_handle	*Delaunay_vertex_handle;*	A type for vertex handles in the Delaunay graph.

Access Methods

Halfedge_handle	*e.twin()*	Returns the twin halfedge.
Halfedge_handle	*e.opposite()*	Same as *e.twin()*.
Halfedge_handle	*e.next()*	Returns the next halfedge in the counterclockwise sense around the boundary of the face that *e* is incident to.
Halfedge_handle	*e.previous()*	Returns the previous halfedge in the counterclockwise sense around the boundary of the adjacent face.

Face_handle	*e.face()*	Returns the face that *e* is incident to.
Vertex_handle	*e.source()*	Returns the source vertex of *e*. *Precondition*: The source vertex must exist, i.e., *has_source()* must return *true*.
Vertex_handle	*e.target()*	Returns the target vertex of *e*. *Precondition*: The target vertex must exist, i.e., *has_target()* must return *true*.
Ccb_halfedge_circulator	*e.ccb()*	Returns a bidirectional circulator to traverse the halfedges on the boundary of the Voronoi face containing *e*. The circulator is initialized to *e*. Applying *operator++* (resp. *operator--*) to this circulator returns the next halfedge on the boundary of the face containing *e* in the counterclockwise (resp. clockwise) sense.
Delaunay_edge	*e.dual()*	Returns the corresponding dual edge in the Delaunay graph.

In the four methods below we consider Voronoi halfedges to be "parallel" to the *x*-axis, oriented from left to right.

Delaunay_vertex_handle	*e.up()*	Returns a handle to the vertex in the Delaunay graph corresponding to the defining site above the Voronoi edge.
Delaunay_vertex_handle	*e.down()*	Returns a handle to the vertex in the Delaunay graph corresponding to the defining site below the Voronoi edge.
Delaunay_vertex_handle	*e.left()*	Returns a handle to the vertex in the Delaunay graph corresponding to the defining site to the left of the Voronoi edge. *Precondition*: *has_source()* must be *true*.
Delaunay_vertex_handle	*e.right()*	Returns a handle to the vertex in the Delaunay graph corresponding to the defining site to the right of the Voronoi edge. *Precondition*: *has_target()* must be *true*.

Predicate Methods

bool	*e.has_source()*	Returns *true* iff the halfedge corresponds to a bisecting segment or a bisecting ray oriented appropriately so that its apex is its source.
bool	*e.has_target()*	Returns *true* iff the halfedge corresponds to a bisecting segment or a bisecting ray oriented appropriately so that its apex is its target.
bool	*e.is_unbounded()*	Returns *true* iff the source or the target of the halfedge does not exist, i.e., if either of *has_source()* or *has_target()* return *false*.
bool	*e.is_bisector()*	Returns *true* iff the Voronoi edge is an entire bisector.
bool	*e.is_segment()*	Returns *true* iff the Voronoi edge has both a source and a target Voronoi vertex.
bool	*e.is_ray()*	Returns *true* iff the Voronoi edge has either a source or a target Voronoi vertex, but not both; in other words it is a bisecting ray.
bool	*e.is_valid()*	Returns *true* if the following conditions are met: the halfedge is not a rejected edge with respect to the chosen adaptation policy; the twin edge of its twin edge is itself; its adjacent face is not a rejected face with respect to the chosen adaptation policy; its source and target vertices are valid (provided they exist, of course); the previous of its next halfedge is itself and the next of its previous halfedge is itself.

See Also

CGAL::Voronoi_diagram_2<DG,AT,AP>
CGAL::Voronoi_diagram_2<DG,AT,AP>::Vertex
CGAL::Voronoi_diagram_2<DG,AT,AP>::Face
DelaunayGraph_2

Class

CGAL::Voronoi_diagram_2<DG,AT,AP>::Vertex

Definition

The class *Vertex* is the Voronoi vertex class provided by the class *Voronoi_diagram_2<DG,AT,AP>* class. Below we present its interface.

Is Model for the Concepts

DefaultConstructible, *CopyConstructible*, *Assignable*, *EqualityComparable*, *LessThanComparable*

Types

Vertex:: Halfedge		A type for the halfedges of the Voronoi diagram.
Vertex:: Face		A type for the faces of the Voronoi diagram.
Vertex:: Vertex_handle		Handle for the vertices of the Voronoi diagram.
Vertex:: Face_handle		Handle for the faces of the Voronoi diagram.
Vertex:: Halfedge_handle		Handle for the halfedges of the Voronoi diagram.
Vertex:: Point_2		A type for the point represented by the vertex.
Vertex:: size_type		A type for sizes.
Vertex:: Halfedge_around_vertex_circulator		A type for a bidirectional circulator that allows to traverse all incident halfedges, i.e., all halfedges that have the vertex as their target. The value type of the circulator is *CGAL::Voronoi_diagram_2< DG,AT,AP>::Halfedge* and is convertible to *Halfedge_handle*.
Vertex:: Delaunay_graph		A type for the Delaunay graph. It is a model of the *DelaunayGraph_2* concept.
typedef Delaunay_graph::Face_handle	*Delaunay_face_handle;*	A type for the handle of the dual face.
typedef Delaunay_graph::Vertex_handle	*Delaunay_vertex_handle;*	A type for the vertex handles in the Delaunay graph.

Access Methods

Halfedge_handle	*v.halfedge()*	Returns an incident halfedge that has *v* as its target.
size_type	*v.degree()*	Returns the in-degree of the vertex, i.e. the number of halfedges that have *v* as their target.
Point_2	*v.point()*	Returns the point represented by the vertex.
Delaunay_face_handle	*v.dual()*	Returns a handle to the corresponding dual face in the Delaunay graph.

Delaunay_vertex_handle	*v.site(unsigned int i)*	Returns a handle to the vertex in the Delaunay graph corresponding to the $(i+1)$-th generating site of the Voronoi vertex. *Precondition*: *i* must be smaller or equal to 2.
Halfedge_around_vertex_circulator	*v.incident_halfedges()*	Returns a bidirectional circulator that allows the traversal of the halfedges that have *v* as their target. Applying *operator++* (resp. *operator--*) to this circulator returns the next incident halfedge in the counterclockwise (resp. clockwise) sense.

Predicate Methods

bool	*v.is_incident_edge(Halfedge_handle e)*	Returns *true* if the halfedge *e* is incident to *v*.
bool	*v.is_incident_face(Face_handle e)*	Returns *true* if the face *f* is incident to *v*.
bool	*v.is_valid()*	Returns *true* if the following conditions are met: the dual face is not an infinite face; all incident halfedges have the vertex as their target.

See Also

CGAL::Voronoi_diagram_2<DG,AT,AP>
CGAL::Voronoi_diagram_2<DG,AT,AP>::Halfedge
CGAL::Voronoi_diagram_2<DG,AT,AP>::Face
DelaunayGraph_2

Class

CGAL::Voronoi_diagram_2<DG,AT,AP>::Face

Definition

The class *Face* is the class provided by the *Voronoi_diagram_2<DG,AT,AP>* class for Voronoi faces. Below we present its interface.

Is Model for the Concepts

DefaultConstructible, *CopyConstructible*, *Assignable*, *EqualityComparable*, *LessThanComparable*

Types

Face:: Vertex	A type for the vertices of the Voronoi diagram.
Face:: Halfedge	A type for the halfedges of the Voronoi diagram.
Face:: Vertex_handle	Handle for the vertices of the Voronoi diagram.
Face:: Face_handle	Handle for the faces of the Voronoi diagram.
Face:: Halfedge_handle	Handle for the halfedges of the Voronoi diagram.
Face:: Ccb_halfedge_circulator	A type for a bidirectional circulator over the halfedges on the boundary of the face. The value type of the circulator is *CGAL::Voronoi_diagram_2<DG,AT,AP>::Halfedge*, and is convertible to *Halfedge_handle*.
Face:: Delaunay_graph	A type for the Delaunay graph. It is a model of the *DelaunayGraph_2* concept.
typedef Delaunay_graph::Vertex_handle Delaunay_vertex_handle;	A type for the handle of the dual vertex.

Access Methods

Halfedge_handle	*f.halfedge()*	Returns an incident halfedge on the boundary of *f*.
Ccb_halfedge_circulator	*f.ccb()*	Returns a bidirectional circulator for traversing the halfedges on the boundary of *f*. The halfedges are traversed in counterclockwise order.
Delaunay_vertex_handle	*f.dual()*	Returns a handle to the corresponding dual vertex in the Delaunay graph.

Predicate Methods

bool	*f.is_unbounded()*	Returns *true* iff the face is an unbounded face in the Voronoi diagram.
bool	*f.is_halfedge_on_ccb(Halfedge e)*	Returns *true* iff *e* is a halfedge of the boundary of *f*.

bool	*f.is_valid()*	Returns *true* iff the following conditions are met: the face is not rejected by the chosen adaptation policy; all its adjacent halfedges do not have zero length; all its adjacent halfedges return the face as their adjacent face.

See Also

CGAL::Voronoi_diagram_2<DG,AT,AP>
CGAL::Voronoi_diagram_2<DG,AT,AP>::Vertex
CGAL::Voronoi_diagram_2<DG,AT,AP>::Halfedge
DelaunayGraph_2

Concept

DelaunayGraph_2

Definition

The concept *DelaunayGraph_2* defines the requirements for the first template parameter of the *Voronoi_diagram_2<DG,AT,AP>* class. The *DelaunayGraph_2* concept essentially defines the requirements that a class representing a Delaunay graph must obey so that the Voronoi diagram adaptor can adapt it.

Refines

DefaultConstructible, *CopyConstructible*, *Assignable*

Types

DelaunayGraph_2:: size_type	A type for sizes.
DelaunayGraph_2:: Geom_traits	A type for the geometric traits associated with the Delaunay graph.
DelaunayGraph_2:: Triangulation_data_structure	A type for the underlying triangulation data structure. It must be a model of the concept *TriangulationDataStructure_2*.
DelaunayGraph_2:: Vertex	A type for the vertices of the Delaunay graph.
DelaunayGraph_2:: Face	A type for the faces of the Delaunay graph.
typedef std::pair<Face_handle,int> Edge;	The type of the edges of the Delaunay graph.
DelaunayGraph_2:: Vertex_handle	Handle to the vertices of the Delaunay graph.
DelaunayGraph_2:: Face_handle	Handle to the faces of the Delaunay graph.

The following iterators and circulators must be defined. All iterators and circulators must be assignable and convertible to their corresponding handles.

DelaunayGraph_2:: All_edges_iterator	A type for an iterator over all edges of the Delaunay graph. Its value type must be *Edge*.
DelaunayGraph_2:: Finite_edges_iterator	A type for an iterator over the finite edges of the Delaunay graph. Its value type must be *Edge*.
DelaunayGraph_2:: All_faces_iterator	A type for an iterator over all faces of the Delaunay graph. Its value type must be *Face*.
DelaunayGraph_2:: Finite_faces_iterator	A type for an iterator over the finite faces of the Delaunay graph. Its value type must be *Face*.
DelaunayGraph_2:: All_vertices_iterator	A type for an iterator over all vertices of the Delaunay graph. Its value type must be *Vertex*.
DelaunayGraph_2:: Finite_vertices_iterator	A type for an iterator over the finite vertices of the Delaunay graph. Its value type must be *Vertex*.
DelaunayGraph_2:: Face_circulator	A type for a circulator over the adjacent faces of a vertex of the Delaunay graph. Its value type must be *Face*.
DelaunayGraph_2:: Vertex_circulator	A type for a circulator over the adjacent vertices of a vertex of the Delaunay graph. Its value type must be *Vertex*.
DelaunayGraph_2:: Edge_circulator	A type for a circulator over the adjacent edges of a vertex of the Delaunay graph. Its value type must be *Edge*.

Creation

In addition to the default and copy constructors, as well as the assignment operator, the following constructors are required.

DelaunayGraph_2 dg(Geom_traits gt);	Constructor that takes an instance of the geometric traits.
template<class It> *DelaunayGraph_2 dg(It first, It beyond);*	Constructor that takes an iterator range. The value type of the iterator must be the type of the sites of the Delaunay graph.
template<class It> *DelaunayGraph_2 dg(It first, It beyond, Geom_traits gt);*	Constructor that takes an iterator range and an instance of the geometric traits. The value type of the iterator must be the type of the sites of the Delaunay graph.

Access methods

Triangulation_data_structure	*dg.tds()*	Returns a reference to the underlying triangulation data structure.
Geom_traits	*dg.geom_traits()*	Returns a reference to the geometric traits object.
Vertex_handle	*dg.infinite_vertex()*	Returns a handle to the infinite vertex.
Vertex_handle	*dg.finite_vertex()*	Returns a handle to a finite vertex, provided there exists one.
Face_handle	*dg.infinite_face()*	Returns a handle to a face incident to the infinite vertex.
int	*dg.dimension()*	Returns the dimension of the Delaunay graph.
size_type	*dg.number_of_vertices()*	Returns the number of finite vertices.
size_type	*dg.number_of_faces()*	Returns the number of faces (both finite and infinite).

Traversal of the Delaunay graph

A model of the *DelaunayGraph_2* concept must provide several iterators and circulators that allow to traverse it (completely or partially). All iterators and circulators must be convertible to the corresponding handles.

Face, Edge and Vertex Iterators

The following iterators must allow, respectively, to visit finite faces, finite edges and finite vertices of the Delaunay graph. These iterators must be non-mutable, bidirectional and their value types are respectively *Face*, *Edge* and *Vertex*.

Finite_vertices_iterator	*dg.finite_vertices_begin()*	Starts at an arbitrary finite vertex.
Finite_vertices_iterator	*dg.finite_vertices_end()*	Past-the-end iterator.
Finite_edges_iterator	*dg.finite_edges_begin()*	Starts at an arbitrary finite edge.
Finite_edges_iterator	*dg.finite_edges_end()*	Past-the-end iterator.
Finite_faces_iterator	*dg.finite_faces_begin()*	Starts at an arbitrary finite face.

Finite_faces_iterator	*dg.finite_faces_end() const*	Past-the-end iterator.

The following iterators must allow, respectively, to visit all (both finite and infinite) faces, edges and vertices of the Delaunay graph. These iterators are non-mutable, bidirectional and their value types are respectively *Face*, *Edge* and *Vertex*.

All_vertices_iterator	*dg.all_vertices_begin()*	Starts at an arbitrary vertex.
All_vertices_iterator	*dg.all_vertices_end()*	Past-the-end iterator.
All_edges_iterator	*dg.all_edges_begin()*	Starts at an arbitrary edge.
All_edges_iterator	*dg.all_edges_end()*	Past-the-end iterator.
All_faces_iterator	*dg.all_faces_begin()*	Starts at an arbitrary face.
All_faces_iterator	*dg.all_faces_end()*	Past-the-end iterator.

Face, Edge and Vertex Circulators

A model of the *DelaunayGraph_2* concept must also provide circulators that allow to visit, respectively, all faces or edges incident to a given vertex or all vertices adjacent to a given vertex. These circulators are non-mutable and bidirectional. The operator *operator++* must move the circulator counterclockwise around the vertex while the *operator--* must move the circulator clockwise.

Face_circulator	*dg.incident_faces(Vertex_handle v)*	Starts at an arbitrary face incident to v.
Face_circulator	*dg.incident_faces(Vertex_handle v, Face_handle f)*	Starts at face f. *Precondition*: Face f must be incident to vertex v.
Edge_circulator	*dg.incident_edges(Vertex_handle v)*	Starts at an arbitrary edge incident to v.
Edge_circulator	*dg.incident_edges(Vertex_handle v, Face_handle f)*	Starts at the first edge of f incident to v, in counterclockwise order around v. *Precondition*: Face f must be incident to vertex v.
Vertex_circulator	*dg.incident_vertices(Vertex_handle v)*	Starts at an arbitrary vertex incident to v.
Vertex_circulator	*dg.incident_vertices(Vertex_handle v, Face_handle f)*	Starts at the first vertex of f adjacent to v in counterclockwise order around v. *Precondition*: Face f must be incident to vertex v.

Predicates

A model of the *DelaunayGraph_2* concept must provide methods to test the finite or infinite character of any feature.

bool	*dg.is_infinite(Vertex_handle v)*	*true*, iff *v* is the *infinite_vertex*.
bool	*dg.is_infinite(Face_handle f)*	*true*, iff face *f* is infinite.
bool	*dg.is_infinite(Face_handle f, int i)*	*true*, iff edge *(f,i)* is infinite.
bool	*dg.is_infinite(Edge e)*	*true*, iff edge *e* is infinite.
bool	*dg.is_infinite(Edge_circulator ec)*	*true*, iff edge **ec* is infinite.

Validity check

bool	*dg.is_valid(bool verbose = false)*	Checks the validity of the Delaunay graph. If *verbose* is *true* a short message is sent to *std::cerr*.

Miscellaneous

void	*dg.clear()*	Clears all contents of the Delaunay graph.
void	*dg.swap(other)*	The Delaunay graphs *other* and *dg* are swapped. *dg.swap(other)* should be preferred to *dg= other* or to *dg(other)* if *other* is deleted afterwards.

Has Models

CGAL::Delaunay_triangulation_2<Traits,Tds>
CGAL::Regular_triangulation_2<Traits,Tds>
CGAL::Triangulation_hierarchy_2<Tr> provided that *Tr* is a model of *DelaunayGraph_2*
CGAL::Segment_Delaunay_graph_2<Gt,DS>
CGAL::Segment_Delaunay_graph_hierarchy_2<Gt,STag,DS>
CGAL::Apollonius_graph_2<Gt,Agds>
CGAL::Apollonius_graph_hierarchy_2<Gt,Agds>

See Also

AdaptationTraits_2
AdaptationPolicy_2
CGAL::Voronoi_diagram_2<DG,AT,AP>

Concept

AdaptationTraits_2

Definition

The concept *AdaptationTraits_2* defines the functors required for accessing geometric information in the Delaunay graph that is needed by the *Voronoi_diagram_2<DG,AT,AP>* class. It optionally defines a functor for performing nearest site queries. A tag is provided for determining whether this functor is defined or not.

Refines

DefaultConstructible, *CopyConstructible*, *Assignable*

Types

AdaptationTraits_2:: Point_2	A type for a point.
AdaptationTraits_2:: Site_2	A type for the sites of the Voronoi diagram.
AdaptationTraits_2:: Delaunay_graph	A type for the triangulated Delaunay graph. The type *Delaunay_graph* must be a model of the *DelaunayGraph_2* concept.

typedef Delaunay_graph::Edge	*Delaunay_edge;*	The type of the edges of the Delaunay graph
typedef Delaunay_graph::Face_handle	*Delaunay_face_handle;*	The type of the face handles of the Delaunay graph
typedef Delaunay_graph::Vertex_handle	*Delaunay_vertex_handle;*	The type of the vertex handles of the Delaunay graph.

AdaptationTraits_2:: Access_site_2

A type for a functor that accesses the site associated with a vertex. The functor should be a model of the concepts *DefaultConstructible*, *CopyConstructible*, *Assignable* and *AdaptableFunctor* (with one argument). The functor must provide the following operator:
result_type operator()(Delaunay_vertex_handle v)
where the result type *result_type* must be either *Site_2* or *const Site_2*`&`.

AdaptationTraits_2:: Construct_Voronoi_point_2

A type for a functor that constructs the dual point of a (triangular) face in the Delaunay graph. This point is the Voronoi vertex of the three sites defining the face in the Delaunay graph. The functor must be a model of the concepts *DefaultConstructible*, *CopyConstructible*, *Assignable*, *AdaptableFunctor* (with one argument). It must provide the following operator:
Point_2 operator()(Delaunay_face_handle f) .
The face handle *f* must not correspond to an infinite face.

AdaptationTraits_2:: Has_nearest_site_2

A tag for determining if the adaptation traits class provides a functor for performing nearest site queries. This tag is equal to either *CGAL::Tag_true* (a nearest site query functor is available) or *CGAL::Tag_false* (a nearest site query functor is not available).

AdaptationTraits_2:: Nearest_site_2

A type for a functor that performs nearest site queries. Semantically, the result of the query is either a face, edge or vertex of the Delaunay graph. It is a face if the query point has at least three closest sites; the returned face has closest sites as vertices. It is an edge if the query point is equidistant to exactly two vertices of the Delaunay graph, which are the source and target vertices of the edge. In all other cases, the search result is a vertex, namely, the unique vertex of the Delaunay graph closest to the query point. The functor must be a model of the concepts *DefaultConstructible*, *CopyConstructible*, *Assignable*, *AdaptableFunctor* (with two arguments). It must provide the following operator:
result_type operator()(Delaunay_graph dg, Point_2 p)
where the result type *result_type* is *boost::variant< Delaunay_vertex_handle,Delaunay_edge,Delaunay_face_handle>*.
This type is required only if *Has_nearest_site_2* is equal to *CGAL::Tag_true*.

Access to objects

Access_site_2	*at.access_site_2_object()*	
Construct_Voronoi_point_2	*at.construct_Voronoi_point_2_object()*	
Nearest_site_2	*at.nearest_site_2_object()*	This method is required only if *Has_nearest_site_2* is equal to *CGAL::Tag_true*.

Has Models

CGAL::Apollonius_graph_adaptation_traits_2<AG2>
CGAL::Delaunay_triangulation_adaptation_traits_2<DT2>
CGAL::Regular_triangulation_adaptation_traits_2<RT2>
CGAL::Segment_Delaunay_graph_adaptation_traits_2<SDG2>

See Also

DelaunayGraph_2
CGAL::Voronoi_diagram_2<DG,AT,AP>

Concept

AdaptationPolicy_2

Definition

The concept *AdaptationPolicy_2* defines the requirements on the predicate functors that determine whether a feature of the triangulated Delaunay graph should be rejected or not. It also provides a functor for inserting sites in the Delaunay graph. The last functor is optional and a tag determines whether it is provided or not. Note that while the first two functors do not modify the Delaunay graph they take as an argument, the last ones does.

Refines

DefaultConstructible, *CopyConstructible*, *Assignable*

Types

AdaptationPolicy_2:: Site_2 — A type for the sites of the Voronoi diagram.

AdaptationPolicy_2:: Delaunay_graph — A type for the triangulated Delaunay graph. The type *Delaunay_graph* must be a model of the *DelaunayGraph_2* concept.

typedef Delaunay_graph::Vertex_handle	*Delaunay_vertex_handle;*
typedef Delaunay_graph::Face_handle	*Delaunay_face_handle;*
typedef Delaunay_graph::Edge	*Delaunay_edge;*
typedef Delaunay_graph::All_edges_iterator	*All_Delaunay_edges_iterator;*
typedef Delaunay_graph::Finite_edges_iterator	*Finite_Delaunay_edges_iterator;*
typedef Delaunay_graph::Edge_circulator	*Delaunay_edge_circulator;*

AdaptationPolicy_2:: Edge_rejector — A type for the predicate functor that is responsible for rejecting an edge of the Delaunay graph (or equivalently rejecting its dual edge in the Voronoi diagram). It must be model of the concepts *DefaultConstructible*, *CopyConstructible*, *Assignable*, and *AdaptableFunctor* (with two arguments). It must provide the following operators:

bool operator()(Delaunay_graph dg, Delaunay_edge e)
bool operator()(Delaunay_graph dg, Delaunay_face_handle f, int i)
bool operator()(Delaunay_graph dg, Delaunay_edge_circulator ec)
bool operator()(Delaunay_graph dg, All_Delaunay_edges_iterator eit)
bool operator()(Delaunay_graph dg, Finite_Delaunay_edges_iterator eit)

The functor returns *true* iff the edge is rejected.

AdaptationPolicy_2:: Face_rejector — A type for the predicate functor that is responsible for rejecting a vertex of the Delaunay graph (or equivalently its dual face in the Voronoi diagram – hence the name of the functor). It must be model of the concepts *DefaultConstructible*, *CopyConstructible*, *Assignable*, *AdaptableFunctor* (with two arguments). It must provide the following operator:

bool operator()(Delaunay graph dg, Delaunay_vertex_handle v)

The functor returns *true* iff the face is rejected.

AdaptationPolicy_2:: Has_inserter — A tag for determining if the adaptation policy class provides a functor for inserting sites in the Delaunay graph. This tag is equal to either *CGAL::Tag_true* (a site inserter functor is available) or *CGAL::Tag_false* (a site inserter functor is not available).

AdaptationPolicy_2:: Site_inserter	A type for a functor that inserts sites in the Delaunay graph. It must be model of the concepts *DefaultConstructible*, *CopyConstructible*, *Assignable*, *AdaptableFunctor* (with two arguments). It must provide the following operator *Delaunay_vertex_handle operator()(Delaunay_graph& dg, Site_2 t)* The vertex handle returned either points to the vertex of the Delaunay graph corresponding to the site just inserted or is the default constructed vertex handle. The latter case can happen if the site inserted is *hidden*, i.e., it has an empty Voronoi cell. This type is required only if the *Has_inserter* tag is equal to *CGAL::Tag_true*.

Access to objects

Edge_rejector	*ap.edge_rejector_object()*	
Face_rejector	*ap.face_rejector_object()*	
Site_inserter	*ap.site_inserter_object()*	This method is required only if *Has_inserter* is equal to *CGAL::Tag_true*.

Miscellaneous

The following methods are important when the adaptation policy maintains a state. This can happen if we have a caching adaptation policy, i.e., when we cache the results of the edge and face rejectors.

void	*ap.clear()*	Clears the state of the adaptation policy.
void	*ap.swap(other)*	The adaptation policies *ap* and *other* are swapped. This method should be preferred to *ap=other* or *ap(other)* if *other* is deleted afterwards.
bool	*ap.is_valid()*	Tests the validity of the adaptation policy.
bool	*ap.is_valid(Delaunay_graph dg)*	Tests the validity of the adaptation policy using extra information from the Delaunay graph *dg*.

Has Models

CGAL::Identity_policy_2<DG,AT>
CGAL::Apollonius_graph_degeneracy_removal_policy_2<AG2>
CGAL::Apollonius_graph_caching_degeneracy_removal_policy_2<AG2>
CGAL::Delaunay_triangulation_degeneracy_removal_policy_2<DT2>
CGAL::Delaunay_triangulation_caching_degeneracy_removal_policy_2<DT2>
CGAL::Regular_triangulation_degeneracy_removal_policy_2<RT2>
CGAL::Regular_triangulation_caching_degeneracy_removal_policy_2<RT2>
CGAL::Segment_Delaunay_graph_degeneracy_removal_policy_2<SDG2>
CGAL::Segment_Delaunay_graph_caching_degeneracy_removal_policy_2<SDG2>

See Also

DelaunayGraph_2
AdaptationTraits_2
CGAL::Voronoi_diagram_2<DG,AT,AP>

Class

CGAL::Apollonius_graph_adaptation_traits_2<AG2>

Definition

The class *Apollonius_graph_adaptation_traits_2<AG2>* provides a model for the *AdaptationTraits_2* concept. The template parameter of the *Apollonius_graph_adaptation_traits_2<AG2>* class must be a model of the *DelaunayGraph_2* concept, and in particular it has the semantics of a (triangulated) 2D Apollonius graph.

#include <CGAL/Apollonius_graph_adaptation_traits_2.h>

Is Model for the Concepts

AdaptationTraits_2

Types

typedef CGAL::Tag_true *Has_nearest_site_2;*

See Also

AdaptationTraits_2
DelaunayGraph_2
CGAL::Voronoi_diagram_2<DG,AT,AP>
CGAL::Apollonius_graph_2<Gt,Agds>
CGAL::Apollonius_graph_hierarchy_2<Gt,Agds>

CGAL::Delaunay_triangulation_adaptation_traits_2<DT2>

Class

Definition

The class *Delaunay_triangulation_adaptation_traits_2<DT2>* provides a model for the *AdaptationTraits_2* concept. The template parameter of the *Delaunay_triangulation_adaptation_traits_2<DT2>* class must be a model of the *DelaunayGraph_2* concept, and in particular it has the semantics of a 2D Delaunay triangulation.

#include <CGAL/Delaunay_triangulation_adaptation_traits_2.h>

Is Model for the Concepts

AdaptationTraits_2

Types

typedef CGAL::Tag_true   *Has_nearest_site_2;*

See Also

AdaptationTraits_2
DelaunayGraph_2
CGAL::Voronoi_diagram_2<DG,AT,AP>
CGAL::Delaunay_triangulation_2<Traits,Tds>

Class

CGAL::Regular_triangulation_adaptation_traits_2<RT2>

Definition

The class *Regular_triangulation_adaptation_traits_2<RT2>* provides a model for the *AdaptationTraits_2* concept. The template parameter of the *Regular_triangulation_adaptation_traits_2<RT2>* class must be a model of the *DelaunayGraph_2* concept, and in particular it has the semantics of a 2D regular triangulation.

#include <CGAL/Regular_triangulation_adaptation_traits_2.h>

Is Model for the Concepts

AdaptationTraits_2

Types

typedef CGAL::Tag_true   *Has_nearest_site_2;*

See Also

AdaptationTraits_2
DelaunayGraph_2
Voronoi_diagram_2<DG,AT,AP>
CGAL::Regular_triangulation_2<Traits,Tds>

CGAL::Segment_Delaunay_graph_adaptation_traits_2<SDG2>

Class

Definition

The class *Segment_Delaunay_graph_adaptation_traits_2<SDG2>* provides a model for the *AdaptationTraits_2* concept. The template parameter of the *Segment_Delaunay_graph_adaptation_traits_2<SDG2>* class must be a model of the *DelaunayGraph_2* concept, and in particular it has the semantics of the 2D (triangulated) segment Delaunay graph.

#include <CGAL/Segment_Delaunay_graph_adaptation_traits_2.h>

Is Model for the Concepts

AdaptationTraits_2

Types

typedef CGAL::Tag_true　　*Has_nearest_site_2;*

See Also

AdaptationTraits_2
DelaunayGraph_2
CGAL::Voronoi_diagram_2<DG,AT,AP>
CGAL::Segment_Delaunay_graph_2<Gt,DS>
CGAL::Segment_Delaunay_graph_hierarchy_2<Gt,STag,DS>

Class

CGAL::Identity_policy_2<DG,AT>

Definition

The class *Identity_policy_2<DG,AT>* provides a model for the *AdaptationPolicy_2* concept. The first template parameter of the *Identity_policy_2<DG,AT>* class must be a model of the *DelaunayGraph_2* concept, whereas as the second template parameter must be a model of the *AdaptationTraits_2* concept. This policy rejects no edge and no face of the Delaunay graph, thus giving a Voronoi diagram which is the true dual of the triangulation Delaunay graph. The Voronoi diagram created with this adaptation policy may have degenerate features, such as Voronoi edges of zero length, or Voronoi faces of zero area. This policy assumes that the Delaunay graph, that is adapted, allows for site insertions through an *insert* method that takes as argument an object of type *AT::Site_2*. The site inserter functor provided by this policy uses the aforementioned *insert* method.

#include <CGAL/Identity_policy_2.h>

Is Model for the Concepts

AdaptationPolicy_2

Types

typedef CGAL::Tag_true   *Has_inserter;*

See Also

AdaptationTraits_2
DelaunayGraph_2
CGAL::Voronoi_diagram_2<DG,AT,AP>

CGAL::Apollonius_graph_degeneracy_removal_policy_2<AG2>

Class

Definition

The class *Apollonius_graph_degeneracy_removal_policy_2<AG2>* provides a model for the *AdaptationPolicy_2* concept. The template parameter of the *Apollonius_graph_degeneracy_removal_policy_2<AG2>* class must be a model of the *DelaunayGraph_2* concept, and in particular it has the semantics of a (triangulated) 2D Apollonius graph. This policy results in a Voronoi diagram that has no degenerate features, i.e., it has no Voronoi edges of zero length and no Voronoi faces of zero area.

#include <CGAL/Apollonius_graph_adaptation_policies_2.h>

Is Model for the Concepts

AdaptationPolicy_2

Types

typedef CGAL::Tag_true   *Has_inserter;*

See Also

AdaptationTraits_2
DelaunayGraph_2
CGAL::Apollonius_graph_caching_degeneracy_removal_policy_2<AG2>
CGAL::Voronoi_diagram_2<DG,AT,AP>
CGAL::Apollonius_graph_2<Gt,Agds>
CGAL::Apollonius_graph_hierarchy_2<Gt,Agds>

Class

CGAL::Delaunay_triangulation_degeneracy_removal_policy_2<DT2>

Definition

The class *Delaunay_triangulation_degeneracy_removal_policy_2<DT2>* provides a model for the *AdaptationPolicy_2* concept. The template parameter of the *Delaunay_triangulation_degeneracy_removal_policy_2<DT2>* class must be a model of the *DelaunayGraph_2* concept, and in particular it has the semantics of a (triangulated) 2D Delaunay triangulation. This policy results in a Voronoi diagram that has no degenerate features, i.e., it has no Voronoi edges of zero length.

#include <CGAL/Delaunay_triangulation_adaptation_policies_2.h>

Is Model for the Concepts

AdaptationPolicy_2

Types

typedef CGAL::Tag_true Has_inserter;

See Also

AdaptationTraits_2
DelaunayGraph_2
CGAL::Delaunay_triangulation_caching_degeneracy_removal_policy_2<DT2>
CGAL::Voronoi_diagram_2<DG,AT,AP>
CGAL::Delaunay_triangulation_2<Traits,Tds>

CGAL::Regular_triangulation_degeneracy_removal_policy_2<RT2>

Class

Definition

The class *Regular_triangulation_degeneracy_removal_policy_2<RT2>* provides a model for the *AdaptationPolicy_2* concept. The template parameter of the *Regular_triangulation_degeneracy_removal_policy_2<RT2>* class must be a model of the *DelaunayGraph_2* concept, and in particular it has the semantics of a (triangulated) 2D regular triangulation. This policy results in a power diagram that has no degenerate features, i.e., it has no Voronoi edges of zero length.

#include <CGAL/Regular_triangulation_adaptation_policies_2.h>

Is Model for the Concepts

AdaptationPolicy_2

Types

typedef CGAL::Tag_true *Has_inserter;*

See Also

AdaptationTraits_2
DelaunayGraph_2
CGAL::Regular_triangulation_caching_degeneracy_removal_policy_2<RT2>
CGAL::Voronoi_diagram_2<DG,AT,AP>
CGAL::Regular_triangulation_2<Traits,Tds>

Class

CGAL::Segment_Delaunay_graph_degeneracy_removal_policy_2< SDG2>

Definition

The class *Segment_Delaunay_graph_degeneracy_removal_policy_2<SDG2>* provides a model for the *AdaptationPolicy_2* concept. The template parameter of the *Segment_Delaunay_graph_degeneracy_removal_policy_2<SDG2>* class must be a model of the *DelaunayGraph_2* concept, and in particular it has the semantics of a (triangulated) 2D segment Delaunay graphs. This policy results in a Voronoi diagram that has no degenerate features, i.e., it has no Voronoi edges of zero length and no Voronoi faces of zero area.

#include <CGAL/Segment_Delaunay_graph_adaptation_policies_2.h>

Is Model for the Concepts

AdaptationPolicy_2

Types

typedef CGAL::Tag_true *Has_inserter;*

See Also

AdaptationTraits_2
DelaunayGraph_2
CGAL::Segment_Delaunay_graph_caching_degeneracy_removal_policy_2<SDG2>
CGAL::Voronoi_diagram_2<DG,AT,AP>
CGAL::Segment_Delaunay_graph_2<Gt,DS>
CGAL::Segment_Delaunay_graph_hierarchy_2<Gt,STag,DS>

CGAL::Apollonius_graph_caching_degeneracy_removal_policy_2< AG2>

Class

Definition

The class *Apollonius_graph_caching_degeneracy_removal_policy_2<AG2>* provides a model for the *AdaptationPolicy_2* concept. The template parameter of the *Apollonius_graph_caching_degeneracy_removal_policy_2<AG2>* class must be a model of the *DelaunayGraph_2* concept, and in particular it has the semantics of a (triangulated) 2D Apollonius graph. This policy caches the results of the edge and face rejectors and results in a Voronoi diagram that has no degenerate features, i.e., no Voronoi edges of zero length and no Voronoi faces of zero area.

#include <CGAL/Apollonius_graph_adaptation_policies_2.h>

Is Model for the Concepts

AdaptationPolicy_2

Types

typedef CGAL::Tag_true *Has_inserter;*

See Also

AdaptationTraits_2
DelaunayGraph_2
CGAL::Apollonius_graph_degeneracy_removal_policy_2<AG2>
CGAL::Voronoi_diagram_2<DG,AT,AP>
CGAL::Apollonius_graph_2<Gt,Agds>
CGAL::Apollonius_graph_hierarchy_2<Gt,Agds>

Class

CGAL::Delaunay_triangulation_caching_degeneracy_removal_policy_2<DT2>

Definition

The class *Delaunay_triangulation_caching_degeneracy_removal_policy_2<DT2>* provides a model for the *AdaptationPolicy_2* concept. The template parameter of the *Delaunay_triangulation_caching_degeneracy_removal_policy_2<DT2>* class must be a model of the *DelaunayGraph_2* concept, and in particular it has the semantics of a (triangulated) 2D Delaunay triangulation. This policy caches the results of the edge and face rejectors and results in a Voronoi diagram that has no degenerate features, i.e., no Voronoi edges of zero length.

#include <CGAL/Delaunay_triangulation_adaptation_policies_2.h>

Is Model for the Concepts

AdaptationPolicy_2

Types

typedef CGAL::Tag_true   *Has_inserter;*

See Also

AdaptationTraits_2
DelaunayGraph_2
CGAL::Delaunay_triangulation_degeneracy_removal_policy_2<DT2>
CGAL::Voronoi_diagram_2<DG,AT,AP>
CGAL::Delaunay_triangulation_2<Traits,Tds>

CGAL::Regular_triangulation_caching_degeneracy_removal_policy_2< RT2>

Class

Definition

The class *Regular_triangulation_caching_degeneracy_removal_policy_2<RT2>* provides a model for the *AdaptationPolicy_2* concept. The template parameter of the *Regular_triangulation_caching_degeneracy_removal_policy_2<RT2>* class must be a model of the *DelaunayGraph_2* concept, and in particular it has the semantics of a (triangulated) 2D regular triangulation. This policy caches the results of the edge and face rejectors and results in a Voronoi diagram that has no degenerate features, i.e., no Voronoi edges of zero length.

#include <CGAL/Regular_triangulation_adaptation_policies_2.h>

Is Model for the Concepts

AdaptationPolicy_2

Types

typedef CGAL::Tag_true   *Has_inserter;*

See Also

AdaptationTraits_2
DelaunayGraph_2
CGAL::Regular_triangulation_degeneracy_removal_policy_2<RT2>
CGAL::Voronoi_diagram_2<DG,AT,AP>
CGAL::Regular_triangulation_2<Traits,Tds>

Class

CGAL::Segment_Delaunay_graph_caching_degeneracy_removal_policy_2<SDG2>

Definition

The class *Segment_Delaunay_graph_caching_degeneracy_removal_policy_2<SDG2>* provides a model for the *AdaptationPolicy_2* concept. The template parameter of the *Segment_Delaunay_graph_caching_degeneracy_removal_policy_2<SDG2>* class must be a model of the *DelaunayGraph_2* concept, and in particular it has the semantics of a (triangulated) 2D segment Delaunay graph. This policy caches the results of the edge and face rejectors and results in a Voronoi diagram that has no degenerate features, i.e., no Voronoi edges of zero length and no Voronoi faces of zero area.

#include <CGAL/Segment_Delaunay_graph_adaptation_policies_2.h>

Is Model for the Concepts

AdaptationPolicy_2

Types

typedef CGAL::Tag_false   Has_inserter;

See Also

AdaptationTraits_2
DelaunayGraph_2
CGAL::Segment_Delaunay_graph_degeneracy_removal_policy_2<SDG2>
CGAL::Voronoi_diagram_2<DG,AT,AP>
CGAL::Segment_Delaunay_graph_2<Gt,DS>
CGAL::Segment_Delaunay_graph_hierarchy_2<Gt,STag,DS>

Part XI

Mesh Generation

Chapter 47

2D Conforming Triangulations and Meshes

Laurent Rineau

Contents

This package implements Shewchuk's algorithm [She00] to construct conforming triangulations and 2D meshes. Conforming triangulations will be described in Section 47.1 and meshes in Section 47.2.

47.1 Conforming Triangulations

47.1.1 Definitions

A triangulation is a *Delaunay triangulation* if the circumscribing circle of any facet of the triangulation contains no vertex in its interior. A *constrained* Delaunay triangulation is a constrained triangulation which is as much Delaunay as possible. The circumscribing circle of any facet of a constrained Delaunay triangulation contains in its interior no data point *visible* from the facet.

An edge is said to be a *Delaunay edge* if it is inscribed in an empty circle (containing no data point in its interior). This edge is said to be a *Gabriel edge* if its diametrical circle is empty.

A constrained Delaunay triangulation is said to be a *conforming Delaunay triangulation* if every constrained edge is a Delaunay edge. Because any edge in a constrained Delaunay triangulation is either a Delaunay edge or a constrained edge, a conforming Delaunay triangulation is in fact a Delaunay triangulation. The only difference is that some of the edges are marked as constrained edges.

A constrained Delaunay triangulation is said to be a *conforming Gabriel triangulation* if every constrained edge is a Gabriel edge. The Gabriel property is stronger than the Delaunay property and each Gabriel edge is a Delaunay edge. Conforming Gabriel triangulations are thus also conforming Delaunay triangulations.

Any constrained Delaunay triangulation can be refined into a conforming Delaunay triangulation or into a conforming Gabriel triangulation by adding vertices, called *Steiner vertices*, on constrained edges until they are decomposed into subconstraints small enough to be Delaunay or Gabriel edges.

47.1.2 Building Conforming Triangulations

Constrained Delaunay triangulations can be refined into conforming triangulations by the two following global functions:
template<class CDT> void make_conforming_Delaunay_2 (CDT& t) and
template<class CDT> void make_conforming_Gabriel_2 (CDT& t).

In both cases, the template parameter *CDT* must be instantiated by a constrained Delaunay triangulation class. Such a class must be a model of the concept *ConstrainedDelaunayTriangulation_2*.

The geometric traits of the constrained Delaunay triangulation used to instantiate the parameter *CDT* has to be a model of the concept *ConformingDelaunayTriangulationTraits_2*.

The constrained Delaunay triangulation *t* is passed by reference and is refined into a conforming Delaunay triangulation or into a conforming Gabriel triangulation by adding vertices. The user is advised to make a copy of the input triangulation in the case where the original triangulation has to be preserved for other computations

The algorithm used by *make_conforming_Delaunay_2* and *make_conforming_Gabriel_2* builds internal data structures that would be computed twice if the two functions are called consecutively on the same triangulation. In order to avoid these data to be constructed twice, the advanced user can use the class *Triangulation_conformer_2<CDT>* to refine a constrained Delaunay triangulation into a conforming Delaunay triangulation and then into a conforming Gabriel triangulation. For additional control of the refinement algorithm, this class also provides separate functions to insert one Steiner point at a time.

47.1.3 Example: Making a Triangulation Conforming Delaunay and Then Conforming Gabriel

This example inserts several segments into a constrained Delaunay triangulation, makes it conforming Delaunay, and then conforming Gabriel. At each step, the number of vertices of the triangulation is printed.

```
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/Constrained_Delaunay_triangulation_2.h>
#include <CGAL/Triangulation_conformer_2.h>

#include <iostream>
```

```
typedef CGAL::Exact_predicates_inexact_constructions_kernel K;
typedef CGAL::Constrained_Delaunay_triangulation_2<K> CDT;
typedef CDT::Point Point;
typedef CDT::Vertex_handle Vertex_handle;

int main()
{
  CDT cdt;

  // construct a constrained triangulation
  Vertex_handle
    va = cdt.insert(Point( 5., 5.)),
    vb = cdt.insert(Point(-5., 5.)),
    vc = cdt.insert(Point( 4., 3.)),
    vd = cdt.insert(Point( 5.,-5.)),
    ve = cdt.insert(Point( 6., 6.)),
    vf = cdt.insert(Point(-6., 6.)),
    vg = cdt.insert(Point(-6.,-6.)),
    vh = cdt.insert(Point( 6.,-6.));

  cdt.insert_constraint(va,vb);
  cdt.insert_constraint(vb,vc);
  cdt.insert_constraint(vc,vd);
  cdt.insert_constraint(vd,va);
  cdt.insert_constraint(ve,vf);
  cdt.insert_constraint(vf,vg);
  cdt.insert_constraint(vg,vh);
  cdt.insert_constraint(vh,ve);

  std::cout << "Number of vertices before: "
            << cdt.number_of_vertices() << std::endl;

  // make it conforming Delaunay
  CGAL::make_conforming_Delaunay_2(cdt);

  std::cout << "Number of vertices after make_conforming_Delaunay_2: "
            << cdt.number_of_vertices() << std::endl;

  // then make it conforming Gabriel
  CGAL::make_conforming_Gabriel_2(cdt);

  std::cout << "Number of vertices after make_conforming_Gabriel_2: "
            << cdt.number_of_vertices() << std::endl;
}
```

File: examples/Mesh_2/conforming.cpp

See figures 47.1, 47.2 and 47.3.

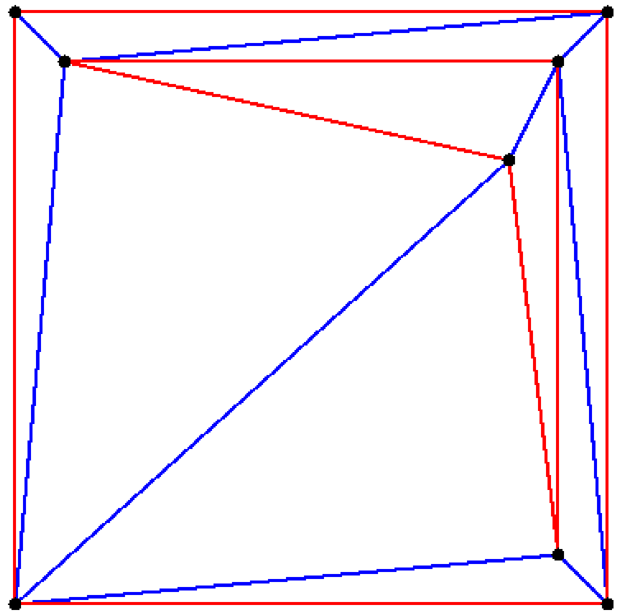


Figure 47.1: Initial triangulation.

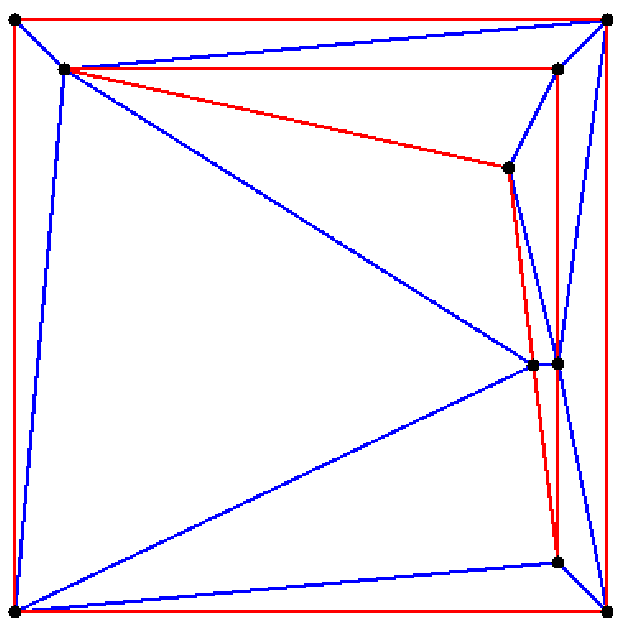


Figure 47.2: The corresponding conforming Delaunay triangulation.

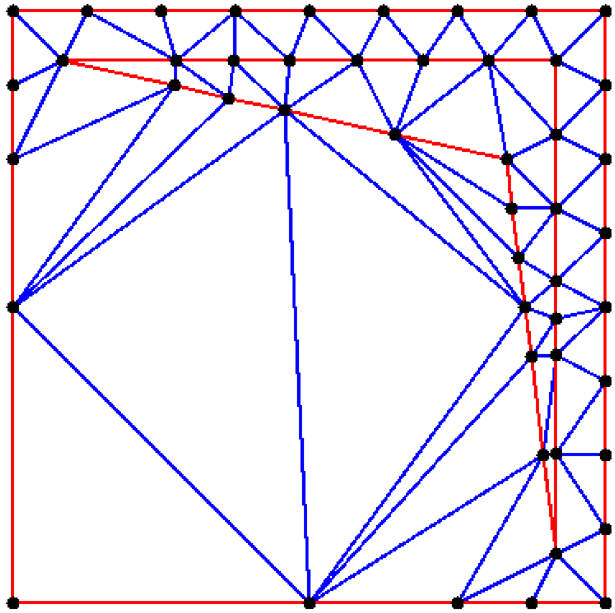


Figure 47.3: The corresponding conforming Gabriel triangulation.

47.2 Meshes

47.2.1 Definitions

A mesh is a partition of a given region into simplices whose shapes and sizes satisfy several criteria.

The domain is the region that the user wants to mesh. It has to be a bounded region of the plane. The domain is defined by a *planar straight line graph*, PSLG for short, which is a set of segments such that two segments in the set are either disjoint or share an endpoint. The segments of the PSLG are constraints that will be represented by a union of edges in the mesh. The PSLG can also contain isolated points that will appear as vertices of the mesh.

The segments of the PSLG are either segments of the boundary or internals constraints. The segments of the PSLG have to cover the boundary of the domain.

The PSLG divides the plane into several connected components. By default, the domain is the union of the bounded connected components. The user can override this default by providing a set of seed points. Either seed points mark components to be meshed or they mark components not to be meshed (holes).

See figures 47.4 and 47.5 for an example of a domain defined without using seed points, and a possible mesh of it. See figure 47.6 for another domain defined with the same PSLG and two seed points used to define holes. In the corresponding mesh (figure 47.7), these two holes are triangulated but not meshed.

47.2.2 Shape and Size Criteria

The shape criterion for triangles is a lower bound B on the ratio between the circumradius and the shortest edge length. Such a bound implies a lower bound of $\arcsin\frac{1}{2B}$ on the minimum angle of the triangle and an upper

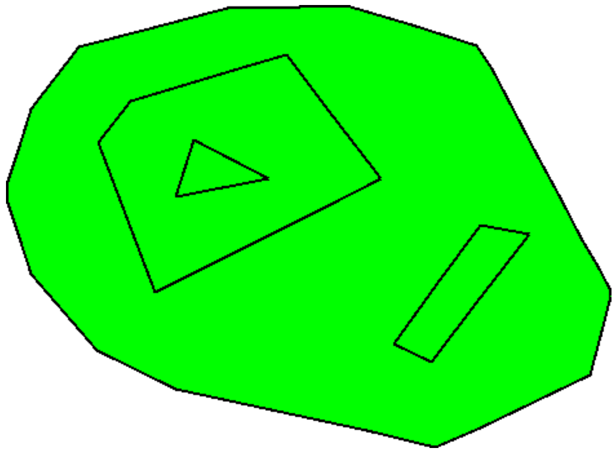


Figure 47.4: A domain defined without seed points.

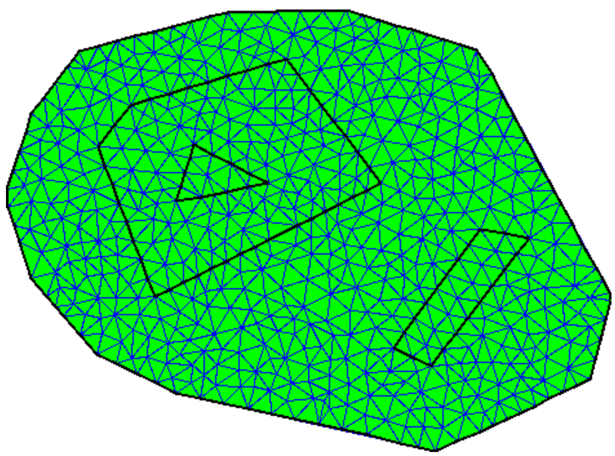


Figure 47.5: A mesh of the domain defined without seed points.

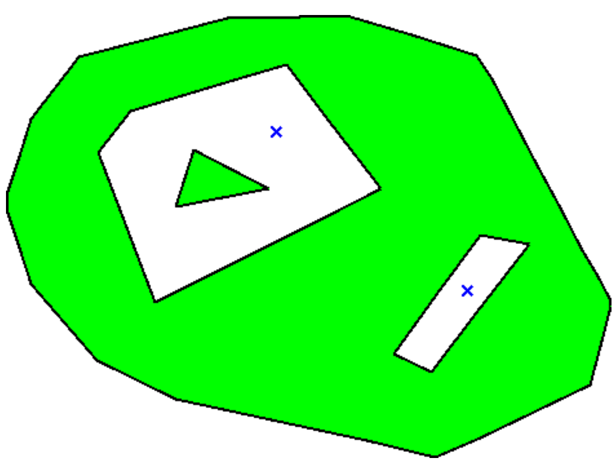


Figure 47.6: A domain with two seeds points defining holes.

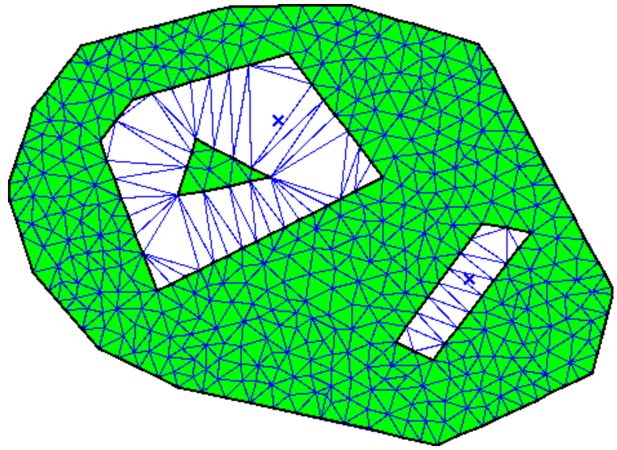


Figure 47.7: A mesh of the domain with two seeds defining holes.

bound of $\pi - 2 * \arcsin \frac{1}{2B}$ on the maximum angle. Unfortunately, the termination of the algorithm is guaranteed only if $B \geq \sqrt{2}$, which corresponds to a lower bound of 20.7 degrees over the angles.

The size criterion can be any criterion that tends to prefer small triangles. For example, the size criterion can be an upper bound on the length of longest edge of triangles, or an upper bound on the radius of the circumcircle. The size bound can vary over the domain. For example, the size criterion could impose a small size for the triangles intersecting a given line.

Both types of criteria are defined in an object *criteria* passed as parameter of the meshing functions.

47.2.3 The Meshing Algorithm

The input to a meshing problem is a PSLG and a set of seeds describing the domain to be meshed, and a set of size and shape criteria. The algorithm implemented in this package starts with a constrained Delaunay triangulation of the input PSLG and produces a mesh using the Delaunay refinement method. This method inserts new vertices to the triangulation, as far as possible from other vertices, and stops when the criteria are satisfied.

If all angles between incident segments of the input PSLG are greater than 60 degrees and if the bound on the circumradius/edge ratio is greater than $\sqrt{2}$, the algorithm is guaranteed to terminate with a mesh satisfying the size and shape criteria.

If some input angles are smaller than 60 degrees, the algorithm will end up with a mesh in which some triangles violate the criteria near small input angles. This is unavoidable since small angles formed by input segments cannot be suppressed. Furthermore, it has been shown ([She00]), that some domains with small input angles cannot be meshed with angles even smaller than the small input angles. Note that if the domain is a polygonal region, the resulting mesh will satisfy size and shape criteria except for the small input angles. In addition, the algorithm may succeed in producing meshes with a lower angle bound greater than 20.7 degrees, but there is no such guarantee.

47.2.4 Building Meshes

Meshes are obtained from constrained Delaunay triangulations by calling the global function
template<class CDT, class Criteria> void refine_Delaunay_mesh_2 (CDT &t, const Criteria& criteria).
The template parameter *CDT* must be instantiated by a constrained Delaunay triangulation class, which is a model of the concept *ConstrainedDelaunayTriangulation_2*. In order to override the domain, a version of this function has two more arguments that define a sequence of seed points.

The geometric traits class of *CDT* has to be a model of the concept *DelaunayMeshTraits_2*. This concept refines the concept *ConformingDelaunayTriangulationTraits_2* adding the geometric predicates and constructors. The template parameter *Criteria* must be a model of *MeshingCriteria_2*. This concept defines criteria that the triangles have to satisfy. CGAL provides two models for this concept:

- *Delaunay_mesh_criteria_2<CDT>*, that defines a shape criterion that bounds the minimum angle of triangles,
- *Delaunay_mesh_size_criteria_2<CDT>*, that adds to the previous criterion a bound on the maximum edge length.

If the function *refine_Delaunay_mesh_2* is called several times on the same triangulation with different criteria, the algorithm rebuilds the internal data structure used for meshing at every call. In order to avoid rebuild the data structure at every call, the advanced user can use the class *Delaunay_mesher_2<CDT>*. This class provides also step by step functions. Those functions insert one vertex at a time.

Any object of type *Delaunay_mesher_2<CDT>* is constructed from a reference to a *CDT*, and has several member functions to define the domain to be meshed and to mesh the *CDT*. See the example given below and the reference manual for details. Note that the *CDT* should not be externally modified during the life time of the *Delaunay_mesher_2<CDT>* object.

Once the mesh is constructed, one can determine which faces of the triangulation are in the mesh domain using the *is_in_domain()* member function of the face type (see the concept *DelaunayMeshFaceBase_2*).

47.2.5 Example Using the Global Function

The following example inserts several segments into a constrained triangulation and then meshes it using the global function *refine_Delaunay_mesh_2*. The size and shape criteria are the default ones provided by the criteria class *Delaunay_mesh_criteria_2<K>*. No seeds are given, meaning that the mesh domain covers the whole plane except the unbounded component.

```
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/Constrained_Delaunay_triangulation_2.h>
#include <CGAL/Delaunay_mesher_2.h>
#include <CGAL/Delaunay_mesh_face_base_2.h>
#include <CGAL/Delaunay_mesh_size_criteria_2.h>

#include <iostream>

typedef CGAL::Exact_predicates_inexact_constructions_kernel K;
typedef CGAL::Triangulation_vertex_base_2<K> Vb;
typedef CGAL::Delaunay_mesh_face_base_2<K> Fb;
typedef CGAL::Triangulation_data_structure_2<Vb, Fb> Tds;
typedef CGAL::Constrained_Delaunay_triangulation_2<K, Tds> CDT;
```

```
typedef CGAL::Delaunay_mesh_size_criteria_2<CDT> Criteria;

typedef CDT::Vertex_handle Vertex_handle;
typedef CDT::Point Point;

int main()
{
  CDT cdt;

  Vertex_handle va = cdt.insert(Point(-4,0));
  Vertex_handle vb = cdt.insert(Point(0,-1));
  Vertex_handle vc = cdt.insert(Point(4,0));
  Vertex_handle vd = cdt.insert(Point(0,1));
  cdt.insert(Point(2, 0.6));

  cdt.insert_constraint(va, vb);
  cdt.insert_constraint(vb, vc);
  cdt.insert_constraint(vc, vd);
  cdt.insert_constraint(vd, va);

  std::cout << "Number of vertices: " << cdt.number_of_vertices() <<
   std::endl;

  std::cout << "Meshing the triangulation..." << std::endl;
  CGAL::refine_Delaunay_mesh_2(cdt, Criteria(0.125, 0.5));

  std::cout << "Number of vertices: " << cdt.number_of_vertices() <<
   std::endl;
}
```

File: examples/Mesh_2/mesh_global.cpp

47.2.6 Example Using the Class Delaunay_mesher_2<CDT>

This example uses the class *Delaunay_mesher_2<CDT>* and calls the *refine_mesh()* member function twice, changing the size and shape criteria in between. In such a case, using twice the global function *refine_Delaunay_mesh_2* would be less efficient, because some internal structures needed by the algorithm would be built twice.

```
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/Constrained_Delaunay_triangulation_2.h>
#include <CGAL/Delaunay_mesher_2.h>
#include <CGAL/Delaunay_mesh_face_base_2.h>
#include <CGAL/Delaunay_mesh_size_criteria_2.h>

#include <iostream>

typedef CGAL::Exact_predicates_inexact_constructions_kernel K;
typedef CGAL::Triangulation_vertex_base_2<K> Vb;
typedef CGAL::Delaunay_mesh_face_base_2<K> Fb;
typedef CGAL::Triangulation_data_structure_2<Vb, Fb> Tds;
typedef CGAL::Constrained_Delaunay_triangulation_2<K, Tds> CDT;
typedef CGAL::Delaunay_mesh_size_criteria_2<CDT> Criteria;
```

```
typedef CGAL::Delaunay_mesher_2<CDT, Criteria> Mesher;

typedef CDT::Vertex_handle Vertex_handle;
typedef CDT::Point Point;

int main()
{
  CDT cdt;

  Vertex_handle va = cdt.insert(Point(-4,0));
  Vertex_handle vb = cdt.insert(Point(0,-1));
  Vertex_handle vc = cdt.insert(Point(4,0));
  Vertex_handle vd = cdt.insert(Point(0,1));
  cdt.insert(Point(2, 0.6));

  cdt.insert_constraint(va, vb);
  cdt.insert_constraint(vb, vc);
  cdt.insert_constraint(vc, vd);
  cdt.insert_constraint(vd, va);

  std::cout << "Number of vertices: " << cdt.number_of_vertices() <<
    std::endl;

  std::cout << "Meshing the triangulation with default criterias..."
            << std::endl;

  Mesher mesher(cdt);
  mesher.refine_mesh();

  std::cout << "Number of vertices: " << cdt.number_of_vertices() <<
    std::endl;

  std::cout << "Meshing with new criterias..." << std::endl;
  // 0.125 is the default shape bound. It corresponds to abound 20.6
    degree.
  // 0.5 is the upper bound on the length of the longuest edge.
  // See reference manual for Delaunay_mesh_size_traits_2<K>.
  mesher.set_criteria(Criteria(0.125, 0.5));
  mesher.refine_mesh();

  std::cout << "Number of vertices: " << cdt.number_of_vertices() <<
    std::endl;
}
```

File: examples/Mesh_2/mesh_class.cpp

47.2.7 Example Using Seeds

This example uses the global function *refine_Delaunay_mesh_2* but defines a domain by using one seed. The size and shape criteria are the default ones provided by the criteria class *Delaunay_mesh_criteria_2<K>*.

Once the mesh is constructed, the *is_in_domain()* member function of faces is used to count them.

```
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/Constrained_Delaunay_triangulation_2.h>
#include <CGAL/Delaunay_mesher_2.h>
#include <CGAL/Delaunay_mesh_face_base_2.h>
#include <CGAL/Delaunay_mesh_size_criteria_2.h>

#include <iostream>

typedef CGAL::Exact_predicates_inexact_constructions_kernel K;
typedef CGAL::Triangulation_vertex_base_2<K> Vb;
typedef CGAL::Delaunay_mesh_face_base_2<K> Fb;
typedef CGAL::Triangulation_data_structure_2<Vb, Fb> Tds;
typedef CGAL::Constrained_Delaunay_triangulation_2<K, Tds> CDT;
typedef CGAL::Delaunay_mesh_size_criteria_2<CDT> Criteria;

typedef CDT::Vertex_handle Vertex_handle;
typedef CDT::Point Point;

int main()
{
  CDT cdt;
  Vertex_handle va = cdt.insert(Point(2,0));
  Vertex_handle vb = cdt.insert(Point(0,2));
  Vertex_handle vc = cdt.insert(Point(-2,0));
  Vertex_handle vd = cdt.insert(Point(0,-2));

  cdt.insert_constraint(va, vb);
  cdt.insert_constraint(vb, vc);
  cdt.insert_constraint(vc, vd);
  cdt.insert_constraint(vd, va);

  va = cdt.insert(Point(3,3));
  vb = cdt.insert(Point(-3,3));
  vc = cdt.insert(Point(-3,-3));
  vd = cdt.insert(Point(3,0-3));

  cdt.insert_constraint(va, vb);
  cdt.insert_constraint(vb, vc);
  cdt.insert_constraint(vc, vd);
  cdt.insert_constraint(vd, va);

  std::list<Point> list_of_seeds;

  list_of_seeds.push_back(Point(0, 0));

  std::cout << "Number of vertices: " << cdt.number_of_vertices() <<
   std::endl;

  std::cout << "Meshing the domain..." << std::endl;
  CGAL::refine_Delaunay_mesh_2(cdt, list_of_seeds.begin(),
   list_of_seeds.end(),
                              Criteria());

  std::cout << "Number of vertices: " << cdt.number_of_vertices() <<
```

```
    std::endl;
  std::cout << "Number of finite faces: " << cdt.number_of_faces() <<
    std::endl;
  int mesh_faces_counter = 0;
  for(CDT::Finite_faces_iterator fit = cdt.finite_faces_begin();
      fit != cdt.finite_faces_end(); ++fit)
  {
    if(fit->is_in_domain()) ++mesh_faces_counter;
  }
  std::cout << "Number of faces in the mesh domain: " <<
    mesh_faces_counter << std::endl;
}
```

File: examples/Mesh_2/mesh_with_seeds.cpp

2D Conforming Triangulations and Meshes Reference Manual

Laurent Rineau

47.3 Classified Reference Pages

Concepts

Classes

Global functions

47.4 Alphabetical List of Reference Pages

ConformingDelaunayTriangulationTraits_2

Concept

Definition

The concept ConformingDelaunayTriangulationTraits_2 refines the concept *ConstrainedDelaunayTriangulationTraits_2* by providing a numeric field type *FT*, a type *Vector_2* and several constructors on *Vector_2*, *Point_2*, and a predicate on angles. The field type has to be a model of the concept *SqrtFieldNumberType*. This field type and the constructors are used by the conforming algorithm to compute Steiner points on constrained edges.

Refines

DelaunayTriangulationTraits_2

Types

ConformingDelaunayTriangulationTraits_2:: FT

The field type. It must be a model of *SqrtFieldNumberType*, that is must be a number type supporting the operations $+$, $-$, $*$, $/$, and $\sqrt{\cdot}$.

ConformingDelaunayTriangulationTraits_2:: Vector_2

The vector type.

ConformingDelaunayTriangulationTraits_2:: Construct_vector_2

Constructor object. Must provide the operator *Vector_2 operator()(Point a, Point b)* that computes the vector $b-a$.

ConformingDelaunayTriangulationTraits_2:: Construct_scaled_vector_2

Constructor object. Must provide the operator *Vector_2 operator()(Vector_2 v, FT scale)* that computes the vector $scale \cdot \mathbf{v}$.

ConformingDelaunayTriangulationTraits_2:: Construct_translated_point_2

Constructor object. Must provide the operator *Point_2 operator()(Point_2 p, Vector_2 v)* that computes the point $p+\mathbf{v}$.

ConformingDelaunayTriangulationTraits_2:: Construct_midpoint_2

Constructor object. Must provide the operator *Point_2 operator()(Point_2 a, Point_2 b)* that computes the midpoint of the segment ab.

ConformingDelaunayTriangulationTraits_2:: Compute_squared_distance_2

Constructor object. Must provide the operator *FT operator()(Point_2 a, Point_2 b)* that computes the squared distance between *a* and *b*.

ConformingDelaunayTriangulationTraits_2:: Angle_2

Predicate object. Must provide the operator *CGAL::Angle operator()(Point_2 p, Point_2 q, Point_2 r)* that returns OBTUSE, RIGHT or ACUTE depending on the angle formed by the three points p, q, r (q being the vertex of the angle).

Access to predicate and constructor objects

Construct_vector_2	*traits.construct_vector_2_object()*
Construct_scaled_vector_2	*traits.construct_scaled_vector_2_object()*
Construct_translated_point_2	*traits.construct_translated_point_2_object()*
Constructor_midpoint_2	*traits.construct_midpoint_2_object()*
Compute_squared_distance_2	*traits.compute_squared_distance_2_object()*
Angle_2	*traits.angle_2_object()*

Has Models

Any model of *Kernel* concept. In particular, all CGAL kernels.

DelaunayMeshFaceBase_2

Definition

The concept DelaunayMeshFaceBase_2 refines the concept *TriangulationFaceBase_2*. It adds two functions giving access to a Boolean marker, that indicates if the face is in the meshing domain or not.

Refines

ConstrainedTriangulationFaceBase_2

Access Functions

bool	*f.is_in_domain() const*	
		returns true if this face is in the domain to be refined.
void	*f.set_in_domain(const bool b)*	
		sets if this face is in the domain.

Has Models

Delaunay_mesh_face_base_2<Traits, Fb>

Concept

Concept

DelaunayMeshTraits_2

Definition

The concept DelaunayMeshTraits_2 refines the concept *ConformingDelaunayTriangulationTraits_2*. It provides a construction object *Construct_circumcenter_2*.

Refines

ConformingDelaunayTriangulationTraits_2

Types

DelaunayMeshTraits_2:: Construct_circumcenter_2	
	Constructor object. Must provide an operator *Point_2 operator()(Point_2 p, Point_2 q, Point_2 r);* that constructs the center of the circle passing through the points p, q, and r. *Precondition*: p, q, and r are not collinear.
DelaunayMeshTraits_2:: Compute_area_2	Constructor object. Must provide an operator *FT operator()(Point_2 p, Point_2 q, Point_2 r);* that computes the signed area of the triangle defined by the points p, q, and r.

Access to predicate and constructor objects

Construct_circumcenter_2	*traits.construct_circumcenter_2_object()*
Compute_area_2	*traits.compute_area_2_object()*

Has Models

Any model of the *Kernel* concept. In particular, all CGAL kernels.

CGAL::Delaunay_mesher_2<CDT, Criteria>

Class

This class implements a 2D mesh generator.

#include <CGAL/Delaunay_mesher_2.h>

Parameters

The template parameter *CDT* should be a model of the concept *ConstrainedDelaunayTriangulation_2*, and type *CDT::Face* should be a model of the concept *MeshFaceBase_2*.

The geometric traits class of the instance of *CDT* has to be a model of the concept *DelaunayMeshTraits_2*.

The template parameter *Criteria* should be a model of the concept *MeshingCriteria_2*. This traits class defines the shape and size criteria for the triangles of the mesh. *Criteria::Face_handle* has to be the same as *CDT::Face_handle*.

Using this class

The constructor of the class *Delaunay_mesher_2<CDT, Criteria>* takes a reference to a *CDT* as an argument. A call to the refinement method *refine_mesh()* will refine the constrained Delaunay triangulation into a mesh satisfying the size and shape criteria specified in the traits class. Note that if, during the life time of the *Delaunay_mesher_2<CDT, Criteria>* object, the triangulation is externally modified, any further call to its member methods may crash. Consider constructing a new *Delaunay_mesher_2<CDT, Criteria>* object if the triangulation has been modified.

Meshing domain

The domain to be mesh is defined by the constrained edges and a set of seed points. The constrained edges divides the plane into several connected components. The mesh domain is either the union of the bounded connected components including at least one seed, or the union of the bounded connected components that do no contain any seed. Note that the unbounded component of the plane is never meshed.

Types

typedef CDT::Geom_traits

Geom_traits; the geometric traits class.

Delaunay_mesher_2<CDT, Criteria>:: Seeds_iterator

const iterator over defined seeds. Its value type is *Geom_traits::Point_2*.

Creation

Delaunay_mesher_2<CDT, Criteria> mesher(CDT& t, Criteria criteria = Criteria());

Create a new mesher, working on *t*, with meshing criteria *criteria*.

Seeds functions

The following functions are used to define seeds.

void	*mesher.clear_seeds()*	Sets seeds to the empty set. All finite connected components of the constrained triangulation will be refined.

template<class InputIterator>
void *mesher.set_seeds(InputIterator begin, InputIterator end, const bool mark=false)*

Sets seeds to the sequence [*begin*, *end*]. If *mark=true*, the mesh domain is the union of the bounded connected components including at least one seed. If *mark=false*, the domain is the union of the bounded components including no seed. Note that the unbounded component of the plane is never meshed.
Requirement: The *value_type* of *begin* and *end* is *Geom_traits::Point_2*.

Seeds_const_iterator *mesher.seeds_begin() const*

Start of the seeds sequence.

Seeds_const_iterator *mesher.seeds_end() const*

Past the end of the seeds sequence.

Meshing methods

void	*mesher.refine_mesh()*	Refines the constrained Delaunay triangulation into a mesh satisfying the criteria defined by the traits.
Criteria	*mesher.get_criteria()*	Returns a const reference to the criteria traits object.

void *mesher.set_criteria(Criteria criteria)*

Assigns *criteria* to the criteria traits object.

advanced

The function *set_criteria* scans all faces to recalculate the list of *bad faces*, that are faces not conforming to the meshing criteria. This function actually has an optional argument that permits to prevent this recalculation. The filling of the list of bad faces can then be done by a call to *set_bad_faces*.

void *mesher.set_criteria(Criteria criteria, bool recalculate_bad_faces)*

Assigns *criteria* to the criteria traits object. If *recalculate_bad_faces* is *false*, the list of bad faces is let empty and the function *set_bad_faces* should be called before *refine_mesh*.

template <class InputIterator>

void *mesher.set_bad_faces(InputIterator begin, InputIterator end)*

This method permits to set the list of bad triangles directly, from the sequence [begin, end], so that the algorithm will not scan the whole set of triangles to find bad ones. To use if there is a non-naive way to find bad triangles.
Requirement: The *value_type* of *begin* and *end* is *Face_handle*.

advanced

advanced

Step by step operations

The *Delaunay_mesher_2<CDT, Criteria>* class allows, for debugging or demos, to play the meshing algorithm step by step, using the following methods.

void *mesher.init()*

This method must be called just before the first call to the following step by step refinement method, that is when all vertices and constrained edges have been inserted into the constrained Delaunay triangulation. It must be called again before any subsequent calls to the step by step refinement method if new vertices or constrained edges have been inserted since the last call.

bool *mesher.is_refinement_done()*

Tests if the step by step refinement algorithm is done. If it returns *true*, the following calls to *step_by_step_refine_mesh* will not insert any points, until some new constrained segments or points are inserted in the triangulation and *init* is called again.

bool *mesher.step_by_step_refine_mesh()*

Applies one step of the algorithm, by inserting one point, if the algorithm is not done. Returns *false* iff no point has been inserted because the algorithm is done.

advanced

Class

CGAL::Delaunay_mesh_criteria_2<CDT>

Definition

The class *Delaunay_mesh_criteria_2<CDT>* is a model for the *MeshingCriteria_2* concept. The shape criterion on triangles is given by a bound B such that for good triangles $\frac{r}{l} \leq B$ where l is the shortest edge length and r is the circumradius of the triangle. By default, $B = \sqrt{2}$, which is the best bound one can use with the guarantee that the refinement algorithm will terminate. The upper bound B is related to a lower bound α_{min} on the minimum angle in the triangle:

$$\sin \alpha_{min} = \frac{1}{2B}$$

so $B = \sqrt{2}$ corresponds to $\alpha_{min} \geq 20.7$ degrees.

#include <CGAL/Delaunay_mesh_criteria_2.h>

Is Model for the Concepts

MeshingCriteria_2

Creation

Delaunay_mesh_criteria_2<CDT> traits; Default constructor. $B = \sqrt{2}$.

Delaunay_mesh_criteria_2<CDT> traits(double b = 0.125);

Construct a traits class with bound $B = \sqrt{\frac{1}{4b}}$.

CGAL::Delaunay_mesh_face_base_2<Traits, Fb>

Class

Definition

The class *Delaunay_mesh_face_base_2<Traits, Fb>* is a model for the concept *DelaunayMeshFaceBase_2*.

This class can be used directly or it can serve as a base to derive other classes with some additional attributes (a color for example) tuned to a specific application.

#include <CGAL/Delaunay_mesh_face_base_2.h>

Parameters

- The first parameter *Traits* is the geometric traits class. It must be the same as the one used for the Delaunay mesh.
- The second parameter *Fb* is the base class from which *Delaunay_mesh_face_base_2<Traits, Fb>* derives. It must be a model of the *TriangulationFaceBase_2* concept.

Inherits From

Fb

Is Model for the Concepts

DelaunayMeshFaceBase_2

Class

CGAL::Delaunay_mesh_size_criteria_2<CDT>

Definition

The class *Delaunay_mesh_size_criteria_2<CDT>* is a model for the *MeshingCriteria_2* concept. The shape criterion on triangles is given by a bound B such that for good triangles $\frac{r}{l} \leq B$ where l is the shortest edge length and r is the circumradius of the triangle. By default, $B = \sqrt{2}$, which is the best bound one can use with the guarantee that the refinement algorithm will terminate. The upper bound B is related to a lower bound α_{min} on the minimum angle in the triangle:

$$\sin \alpha_{min} = \frac{1}{2B}$$

so $B = \sqrt{2}$ corresponds to $\alpha_{min} \geq 20.7$ degrees.

This traits class defines also a size criteria: all segments of all triangles must be shorter than a bound S.

#include <CGAL/Delaunay_mesh_size_criteria_2.h>

Is Model for the Concepts

MeshingCriteria_2

Creation

Delaunay_mesh_size_criteria_2<CDT> traits; Default constructor. $B = \sqrt{2}$. No bound on size

Delaunay_mesh_size_criteria_2<CDT> traits(double b = 0.125, double S = 0);

Construct a traits class with bound $B = \sqrt{\frac{1}{4b}}$. If $S \neq 0$, the size bound is S. If $S = 0$, there is no bound on size.

CGAL::Mesh_2::Face_badness

#include <CGAL/Mesh_2/Face_badness.h>

enum Mesh_2::Face_badness { *NOT_BAD, BAD, IMPERATIVELY_BAD*}*;*

Function

CGAL::make_conforming_Delaunay_2

#include <CGAL/Triangulation_conformer_2.h>

template<class CDT>
void *make_conforming_Delaunay_2(CDT &t)*

Refines the constrained Delaunay triangulation *t* into a conforming Delaunay triangulation. After a call to this function, all edges of *t* are Delaunay edges.
Requirement: The template parameter *CDT* should be a model of the concept *ConstrainedDelaunayTriangulation_2*. The geometric traits class of into the constrained Delaunay triangulation must be a model of *ConformingDelaunayTriangulationTraits_2*.

Function

CGAL::make_conforming_Gabriel_2

#include <CGAL/Triangulation_conformer_2.h>

template<class CDT>
void *make_conforming_Gabriel_2(CDT &t)*

Refines the constrained Delaunay triangulation *t* into a conforming Gabriel triangulation. After a call to this function, all constrained edges of *t* have the *Gabriel property*: the circle that has *e* as diameter does not contain any vertex from the triangulation.
Requirement: The template parameter *CDT* should be a model of the concept *ConstrainedDelaunayTriangulation_2*. The geometric traits class of the constrained Delaunay triangulation must be a model of *ConformingDelaunayTriangulationTraits_2*.

Concept

MeshingCriteria_2

Definition

The concept MeshingCriteria_2 defines the meshing criteria to be used in the algorithm. It provides a predicate *Is_bad* that tests a triangle according to criteria. The return type of *Is_bad* is an enum *Mesh_2::Face_badness*.

The possible values of *Mesh_2::Face_badness* are *NOT_BAD*, *BAD* and *IMPERATIVELY_BAD*. If the predicate returns *BAD*, the triangle is marked as bad and the algorithm will try to destroy it. If the predicates returns *IMPERATIVELY_BAD*, the algorithm will destroy the triangle unconditionally during its execution.

The termination of the algorithm is guaranteed when criteria are shape criteria corresponding to a bound on smallest angles not less than 20.7 degrees (this corresponds to a radius-edge ratio bound not less than $\sqrt{2}$). Any size criteria that are satisfied by small enough triangle can be added to the set of criteria without compromising the termination.

Note that, in the presence of input angles smaller than 60 degrees, some bad shaped triangles can appear in the final mesh in the neighboring of small angles. To achieve termination and the respect of size criteria everywhere, the *Is_bad* predicate has to return *IMPERATIVELY_BAD* when size criteria are not satisfied, and *BAD* when shape criteria are not satisfied.

MeshingCriteria_2 also provides a type *Quality* designed to code a quality measure for triangles. The type *Quality* must be *less-than comparable* as the meshing algorithm will order bad triangles by quality, to split those with smallest quality first. The predicate *Is_bad* computes the quality of the triangle as a by-product.

Types

MeshingCriteria_2:: Face_handle	Handle to a face of the triangulation.
MeshingCriteria_2:: Quality	Default constructible, copy constructible, assignable, and less-than comparable type.
MeshingCriteria_2:: Is_bad	Predicate object. Must provide two operators. The first operator *Mesh_2::Face_badness operator()(Face_handle fh, Quality& q)* returns *NOT_BAD* if it satisfies the desired criteria for mesh triangles, *BAD* if it does not, and *IMPERATIVELY_BAD* if it does not and should be refined unconditionally. In addition, this operator assigns to q a value measuring the quality of the triangle pointed by *fh*. The second operator *Mesh_2::Face_badness operator()(Quality q)* returns *NOT_BAD* if q is the quality of a good triangle, *BAD* if the q represents a poor quality, and *IMPERATIVELY_BAD* if q represents the quality of a bad triangle that should be refined unconditionally.

Access to predicate and constructor objects

Is_bad *traits.is_bad_object()*

Has Models

Delaunay_mesh_criteria_2<CDT>
Delaunay_mesh_size_criteria_2<CDT>

Function

CGAL::refine_Delaunay_mesh_2

template<class CDT, class Criteria>
void *refine_Delaunay_mesh_2(CDT &t, Criteria criteria = Criteria())*

Refines the default domain defined by a constrained Delaunay triangulation without seeds into a mesh satisfying the criteria defined by the traits *criteria*. The domain of the mesh covers all the connected components of the plane defined by the constrained edges of *t*, except for the unbounded component.

Precondition: The template parameter *CDT* must be a model of the concept *ConstrainedDelaunayTriangulation_2*. The geometric traits class of the constrained Delaunay triangulation must be a model of *DelaunayMeshTraits_2*.

Requirement: The face of the constrained Delaunay triangulation must be a model of the concept *DelaunayMeshFaceBase_2*. *Criteria* must be a model of the concept *MeshingCriteria_2* and *CDT::Face_handle* must be the same as *Criteria::Face_handle*.

template <class CDT, class Criteria, class InputIterator>
void *refine_Delaunay_mesh_2(CDT& t,*
InputIterator begin,
InputIterator end,
Criteria criteria = Criteria(),
bool mark = false)

Refines the default domain defined by a constrained Delaunay triangulation into a mesh satisfying the criteria defined by the traits *criteria*.The sequence [*begin*, *end*] gives a set of seeds points, that defines the domain to be meshed as follows. The constrained edges of *t* partition the plane into connected components. If *mark=true*, the mesh domain is the union of the bounded connected components including at least one seed. If *mark=false*, the domain is the union of the bounded components including no seed. Note that the unbounded component of the plane is never meshed.

Requirement: The *value_type* of *begin* and *end* is *CDT::Geom_traits::Point_2*.

CGAL::Triangulation_conformer_2<CDT>

Class

The class *Triangulation_conformer_2<CDT>* is an auxiliary class of *Delaunay_mesher_2<CDT>*. It permits to refine a constrained Delaunay triangulation into a conforming Delaunay or conforming Gabriel triangulation. For standard needs, consider using the global functions *make_conforming_Gabriel_2* and *make_conforming_Delaunay_2*.

#include <CGAL/Triangulation_conformer_2.h>

Parameters

The template parameter *CDT* should be a model of the concept *ConstrainedDelaunayTriangulation_2*.

The geometric traits class of the instance of *CDT* has to be a model of the concept *ConformingDelaunayTriangulationTraits_2*.

Using this class

The constructor of the class *Triangulation_conformer_2<CDT>* takes a reference to a *CDT* as an argument. A call to the method *make_conforming_Delaunay()* or *make_conforming_Gabriel()* will refine this constrained Delaunay triangulation into a conforming Delaunay or conforming Gabriel triangulation. Note that if, during the life time of the *Triangulation_conformer_2<CDT>* object, the triangulation is externally modified, any further call to its member methods may lead to undefined behavior. Consider reconstructing a new *Triangulation_conformer_2<CDT>* object if the triangulation has been modified.

The conforming methods insert points into constrained edges, thereby splitting them into several subconstraints. You have access to the initial inserted constraints if you instantiate the template parameter by a *CGAL::Constrained_triangulation_plus_2<CDT>*.

Creation

Triangulation_conformer_2<CDT> m(CDT& t);

Create a new conforming maker, working on *t*.

Operations

Conforming methods

void	*m.make_conforming_Delaunay()*	Refines the triangulation into a conforming Delaunay triangulation. After a call to this method, all triangles fulfill the Delaunay property, that is the empty circle property.

void *m.make_conforming_Gabriel()*

Refines the triangulation into a conforming Gabriel triangulation. After a call to this method, all constrained edges *e* have the *Gabriel property*: the circle with diameter *e* does not contain any vertex of the triangulation.

Checking

The following methods verify that the constrained triangulation is conforming Delaunay or conforming Gabriel. These methods scan the whole triangulation and their complexity is proportional to the number of edges.

bool *m.is_conforming_Delaunay()*

Returns *true* iff all triangles fulfill the Delaunay property.

bool *m.is_conforming_Gabriel()*

Returns *true* iff all constrained edges have the Gabriel property: their circumsphere is empty.

advanced

Step by step operations

The *Triangulation_conformer_2<CDT>* class allows, for debugging or demos, to play the conforming algorithm step by step, using the following methods. They exist in two versions, depending on whether you want the triangulation to be conforming Delaunay or conforming Gabriel, respectively. Any call to a *step_by_step_conforming_XX* function requires a previous call to the corresponding function *init_XX* and Gabriel and Delaunay methods can not be mixed between two calls of *init_XX*.

void *m.init_Delaunay()* The method must be called after all points and constrained segments are inserted and before any call to the following methods. If some points or segments are then inserted in the triangulation, this method must be called again.

bool *m.step_by_step_conforming_Delaunay()*

Applies one step of the algorithm, by inserting one point, if the algorithm is not done. Returns *false* iff no point has been inserted because the algorithm is done.

void *m.init_Gabriel()* Analog to *init_Delaunay* for Gabriel conforming.

bool *m.step_by_step_conforming_Gabriel()*

Analog to *step_by_step_conforming_Delaunay()* for Gabriel conforming.

bool *m.is_conforming_done()*

Tests if the step by step conforming algorithm is done. If it returns *true*, the following calls to *step_by_step_conforming_XX* will not insert any points, until some new constrained segments or points are inserted in the triangulation and *init_XX* is called again.

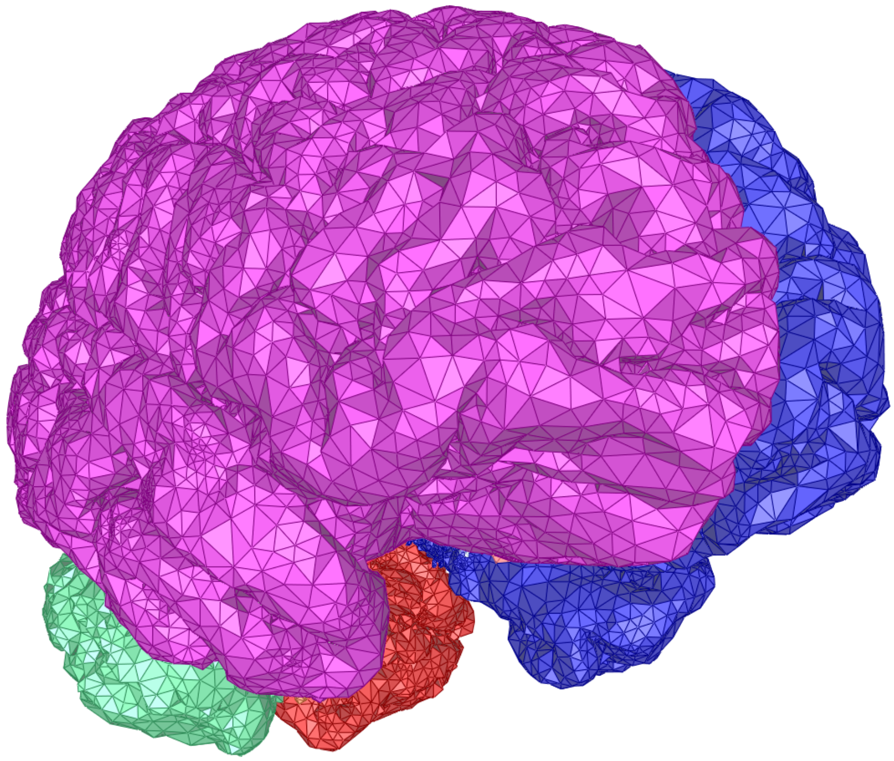
advanced

Chapter 48

3D Surface Mesh Generation

Laurent Rineau and Mariette Yvinec

Contents



48.1 Introduction

This package provides a function template to compute a triangular mesh approximating a surface.

The meshing algorithm requires to know the surface to be meshed only through an oracle able to tell whether a given segment, line or ray intersects the surface or not and to compute an intersection point if any. This feature makes the package generic enough to be applied in a wide variety of situations. For instance, it can be used to mesh implicit surfaces described as the zero level set of some function. It may also be used in the field of medical imaging to mesh surfaces described as a gray level set in a three dimensional image.

The meshing algorithm is based on the notion of the restricted Delaunay triangulation. Basically the algorithm computes a set of sample points on the surface, and extract an interpolating surface mesh from the three dimensional triangulation of these sample points. Points are iteratively added to the sample, as in a Delaunay refinement process, until some size and shape criteria on the elements of the surface mesh are satisfied.

The size and shape criteria guide the behavior of the refinement process and control its termination. They also condition the size and shape of the elements in the final mesh. Naturally, those criteria can be customized to satisfy the user needs. The *Surface mesh generation* package offers a set of standard criteria that can be scaled through three numerical values. Also the user can also plug in its own set of refinement criteria.

There is no restriction on the topology and number of components of the surface provided that the oracle (or the user) is able to provide one initial sample point on each connected component. If the surface is smooth enough, and if the size criteria are small enough, the algorithm guarantees that the output mesh is homeomorphic to the surface, and is within a small bounded distance (Hausdorff or even Frechet distance) from the surface. The algorithm can also be used for non smooth surfaces but then there is no guarantee.

48.2 The Surface Mesh Generator Interface for Smooth Surfaces

The meshing process is launched through a call to a function template. There are two overloaded versions of the meshing function whose signatures are the following:

template <class SurfaceMeshC2T3, class Surface, class FacetsCriteria, class Tag >
void make_surface_mesh(
SurfaceMeshC2T3& c2t3,
Surface surface,
FacetsCriteria criteria,
Tag)

template <class SurfaceMeshC2T3, class SurfaceMeshTraits, class FacetsCriteria, class Tag >
void make_surface_mesh(
SurfaceMeshC2T3& c2t3,
SurfaceMeshTraits::Surface_3 surface,
SurfaceMeshTraits traits,
FacetsCriteria criteria,
Tag)

The template parameter *SurfaceMeshC2T3* stands for a data structure type that is used to store the surface mesh. This type is required to be a model of the concept *SurfaceMeshComplex_2InTriangulation_3*. Such a data structure has a pointer to a three dimensional triangulation and encodes the surface mesh as a subset of facets in this triangulation. An argument of type *SurfaceMeshC2T3* is passed by reference to the meshing function. This argument holds the output mesh at the end of the process.

The template parameter *Surface* stands for the surface type. This type has to be a model of the concept *Surface_3*.

The knowledge on the surface, required by the surface mesh generator is encapsulated in a traits class. Actually, the mesh generator accesses the surface to be meshed through this traits class only. The traits class is required to be a model of the concept *SurfaceMeshTraits_3*. The difference between the two overloaded versions of *make_surface_mesh* can be explained as follows

- In the first overloaded version of *make_surface_mesh*, the surface type is given as template parameter (*Surface*) and the *surface* to be meshed is passed as parameter to the mesh generator. In that case the surface mesh generator traits type is automatically generated from the surface type by an auxiliary class called the *Surface_mesh_traits_generator_3*.
- In the second overloaded version of *make_surface_mesh*, the surface mesh generator traits type is provided by the template parameter *SurfaceMeshTraits_3* and the surface type is obtained from this traits type. Both a surface and a traits are passed to the mesh generator as arguments.

The first overloaded version can be used whenever the surface type either provides a nested type *Surface::Surface_mesher_traits_3* that is a model of *SurfaceMeshTraits_3* or is a surface type for which a specialization of the traits generator *Surface_mesh_traits_generator_3<Surface>* is provided. Currently, the library provides partial specializations of *Surface_mesher_traits_generator_3<Surface>* for implicit surfaces (*Implicit_surface_3<Traits, Function>*) and gray level images (*Gray_level_image_3<FT, Point>*).

The parameter *criteria* handles the description of the size and shape criteria driving the meshing process. The template parameter *FacetsCriteria* has to be instantiated by a model of the concept *SurfaceMeshFacetsCriteria*.

The parameter *Tag* is a tag whose type influences the behavior of the meshing algorithm. For instance, this parameter can be used to enforce the manifold property of the output mesh while avoiding an over-refinement of the mesh. Further details on this subject are given in Section 48.4.

A call to *make_surface_mesh(c2t3,surface, criteria, tag)* launches the meshing process with an initial set of points which is the union of two subsets: the set of vertices in the initial triangulation pointed to by *c2t3*, and a set of points provided by the *Construct_initial_points()* functor of the traits class. This initial set of points is required to include at least one point on each connected component of the surface to be meshed.

48.3 Examples

48.3.1 Meshing Isosurfaces defined by Implicit Functions

The first code example meshes a sphere given as the zero level set of a function $\mathbb{R}^3 \longrightarrow \mathbb{R}$. More precisely, the surface to be meshed is created by the constructor of the class *Implicit_surface_3<Kernel, Function>* from a pointer to the function (*sphere_function*) and a bounding sphere.

The default meshing criteria are determined by three numerical values:

- *angular_bound* is a lower bound in degrees for the angles of mesh facets.
- *radius_bound* is an upper bound on the radii of surface Delaunay balls. A surface Delaunay ball is a ball circumscribing a mesh facet and centered on the surface.
- *distance_bound* is an upper bound for the distance between the circumcenter of a mesh facet and the center of a surface Delaunay ball of this facet.

Given this surface type, the surface mesh generator will use an automatically generated traits class.

The resulting mesh is shown in Figure 48.1.

```
#include <CGAL/Surface_mesh_default_triangulation_3.h>
#include <CGAL/Complex_2_in_triangulation_3.h>
#include <CGAL/make_surface_mesh.h>
#include <CGAL/Implicit_surface_3.h>

// default triangulation for Surface_mesher
typedef CGAL::Surface_mesh_default_triangulation_3 Tr;

// c2t3
typedef CGAL::Complex_2_in_triangulation_3<Tr> C2t3;

typedef Tr::Geom_traits GT;
typedef GT::Sphere_3 Sphere_3;
typedef GT::Point_3 Point_3;
typedef GT::FT FT;

typedef FT (*Function)(Point_3);

typedef CGAL::Implicit_surface_3<GT, Function> Surface_3;

FT sphere_function (Point_3 p) {
```

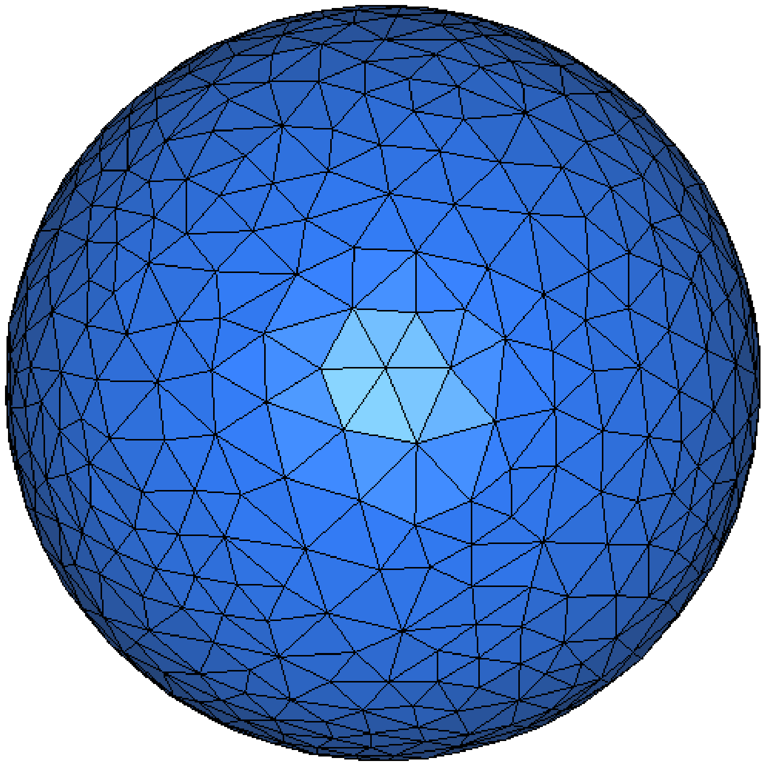


Figure 48.1: Surface mesh of a sphere

```
  const FT x2=p.x()*p.x(), y2=p.y()*p.y(), z2=p.z()*p.z();
  return x2+y2+z2-1;
}

int main() {
  Tr tr;            // 3D-Delaunay triangulation
  C2t3 c2t3 (tr);   // 2D-complex in 3D-Delaunay triangulation

  // defining the surface
  Surface_3 surface(sphere_function,             // pointer to function
                    Sphere_3(CGAL::ORIGIN, 2.)); // bounding sphere
  // Note that "2." above is the *squared* radius of the bounding sphere!

  // defining meshing criteria
  CGAL::Surface_mesh_default_criteria_3<Tr> criteria(30.,  // angular
   bound
                                                     0.1,  // radius
   bound
                                                     0.1); // distance
   bound
  // meshing surface
  CGAL::make_surface_mesh(c2t3, surface, criteria,
   CGAL::Non_manifold_tag());

  std::cout << "Final number of points: " << tr.number_of_vertices() <<
   "\n";
```

```
}
```

File: examples/Surface_mesher/mesh_an_implicit_function.cpp

48.3.2 Meshing Isosurfaces Defined as Gray Levels in 3D Images

In this example the surface to be meshed is defined as the locus of points with a given gray level in a 3D image. The code is quite similar to the previous example.

The main difference with the previous code is that the function used to define the surface is an object of type *CGAL::Gray_level_image_3* created from an image file and a numerical value that is the gray value of the level one wishes to mesh.

Note that surface, which is still an object of type *Implicit_surface_3* is now, defined by three parameters that are the function, the bounding sphere and a numerical value called *the precision*. This precision, whose value is relative to the bounding sphere radius, is used in the intersection computation. This parameter has a default which was used in the previous example. Also note that the center of the bounding sphere is required to be internal a point where the function has a negative value.

The chosen iso-value of this 3D image corresponds to a head skull. The resulting mesh is shown in Figure 48.2.

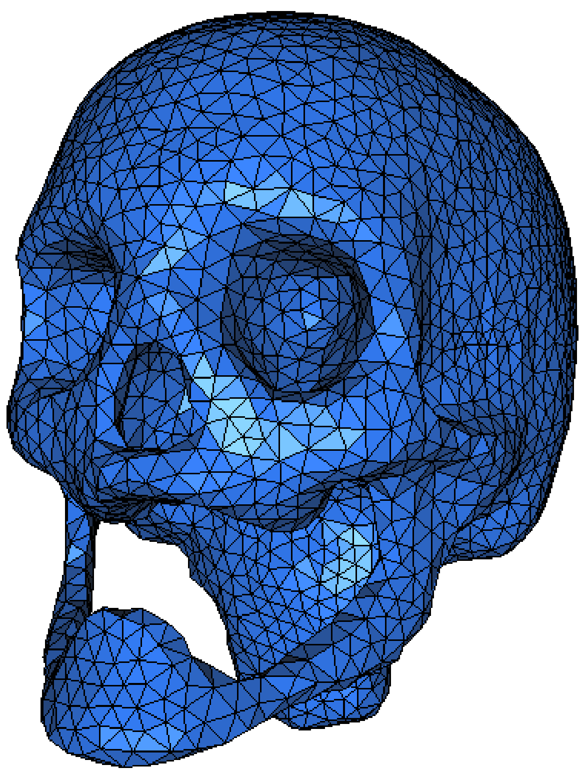


Figure 48.2: Surface mesh of an iso-contour extracted from a gray level 3D image

```
#include <CGAL/Surface_mesh_default_triangulation_3.h>
#include <CGAL/Surface_mesh_default_criteria_3.h>
#include <CGAL/Complex_2_in_triangulation_3.h>
```

```
#include <CGAL/IO/Complex_2_in_triangulation_3_file_writer.h>
#include <fstream>
#include <CGAL/make_surface_mesh.h>
#include <CGAL/Gray_level_image_3.h>
#include <CGAL/Implicit_surface_3.h>

// default triangulation for Surface_mesher
typedef CGAL::Surface_mesh_default_triangulation_3 Tr;

// c2t3
typedef CGAL::Complex_2_in_triangulation_3<Tr> C2t3;

typedef Tr::Geom_traits GT;
typedef CGAL::Gray_level_image_3<GT::FT, GT::Point_3> Gray_level_image;
typedef CGAL::Implicit_surface_3<GT, Gray_level_image> Surface_3;

int main() {
  Tr tr;            // 3D-Delaunay triangulation
  C2t3 c2t3 (tr);   // 2D-complex in 3D-Delaunay triangulation

  // the 'function' is a 3D gray level image
  Gray_level_image image("data/skull_2.9.inr", 2.9f);

  // Carefully choosen bounding sphere: the center must be inside the
  // surface defined by 'image' and the radius must be high enough so
   that
  // the sphere actually bounds the whole image.
  GT::Point_3 bounding_sphere_center(122., 102., 117.);
  GT::FT bounding_sphere_squared_radius = 200.*200.*2.;
  GT::Sphere_3 bounding_sphere(bounding_sphere_center,
                                   bounding_sphere_squared_radius);

  // definition of the surface, with 10^-5 as relative precision
  Surface_3 surface(image, bounding_sphere, 1e-5);

  // defining meshing criteria
  CGAL::Surface_mesh_default_criteria_3<Tr> criteria(30.,
                                                      5.,
                                                      5.);

  // meshing surface, with the "manifold without boundary" algorithm
  CGAL::make_surface_mesh(c2t3, surface, criteria, CGAL::Manifold_tag());
  std::ofstream out("out.off");
  CGAL::output_surface_facets_to_off (out, c2t3);
  std::cout << "Final number of points: " << tr.number_of_vertices() <<
   "\n";
}
```

File: examples/Surface_mesher/mesh_a_3d_gray_image.cpp

48.4 Meshing Criteria, Guarantees and Variations

The guarantees on the output mesh depend on the mesh criteria. Theoretical guarantees are given in [BO05]. First, the meshing algorithm is proved to terminate if the lower bound on facets angles is not bigger than 30 degrees. Furthermore, the output mesh is guaranteed to be homeomorphic to the surface, and there is a guaranteed bound on the distance (Hausdorff and even Frechet distance) between the mesh and the surface if the radius bound is everywhere smaller than ε times the local feature size. Here ε is a constant that has to be less than 0.16, and the local feature size $lfs(x)$ is defined on each point x of the surface as the distance from x to the medial axis. Note that the radius bound need not be uniform, although it is a uniform bound in the default criteria.

Naturally, such a theoretical guarantee can be only achieved for smooth surfaces that have a finite, non zero reach value. (The reach of a surface is the minimum value of local feature size on this surface).

The value of the local feature size on any point of the surface or its minimum on the surface it usually unknown although it can sometimes be guessed. Also it happens frequently that setting the meshing criteria so as to fulfill the theoretical conditions yields an over refined mesh. On the other hand, when the size criteria are relaxed, no homeomorphism with the input surface is guaranteed, and the output mesh is not even guaranteed to be manifold. To remedy this problem and give a more flexible meshing algorithm, the function template *make_surface_mesh* has a tag template parameter allowing to slightly change the behavior of the refinement process. This feature allows, for instance, to run the meshing algorithm with a relaxed size criteria, more coherent with the size of the mesh expected by the user, and still have a guarantee that the output mesh forms a manifold surface. The function *make_surface_mesh* has specialized versions for the following tag types:
Manifold_tag: the output mesh is guaranteed to be a manifold surface without boundary.
Manifold_with_boundary_tag: the output mesh is guaranteed to be manifold but may have boundaries.
Non_manifold_tag: the algorithm relies on the given criteria and guarantees nothing else.

48.5 Output

This CGAL component also provides functions to write the reconstructed surface mesh to the Object File Format (OFF) [Phi96] and to convert it to a polyhedron (when it is manifold):
CGAL::output_surface_facets_to_off
CGAL::output_surface_facets_to_polyhedron

48.6 Undocumented Features Available in Demos

The Polyhedron demo has a feature that allows to remesh a polyhedral surface, using the 3D Surface Mesh Generator. That has been implemented as a special model of *SurfaceMeshTraits_3*, for polyhedra. That traits class is not yet documented because its interface and code have not yet been stabilized. It will probably be shipped with next release of CGAL.

The Surface Mesh Generator demo allows to mesh not only gray level images, but also segmented images, when voxels are labelled with a domain index. Such images are for example the result of a segmentation of 3D medical images. This feature is not yet ready to be documented in current release, but will probably be in next release of CGAL.

48.7 Design and Implementation History

The algorithm implemented in this package is mainly based on the work of Jean-Daniel Boissonnat and Steve Oudot [BO05]. Steve Oudot implemented a first working prototype of the algorithm during his PhD thesis.

The meshing algorithm is implemented using the design of mesher levels described in [RY07].

David Rey, Steve Oudot and Andreas Fabri have participated in the development of this package.

3D Surface Mesh Generation Reference Manual

Laurent Rineau and Mariette Yvinec

The surface mesh generation package offers a function template which builds a triangular mesh approximating a surface.

The meshing algorithms requires to know the surface to be meshed through an oracle that mainly can tell whether a given segment, ray or line intersects the surface or not and can compute the compute the intersections point if any. The oracle is represented by a traits class which can be passed to the meshing function or automatically generated for certain types of surfaces. The current implementation provides traits classes to mesh implicit surfaces as well as surfaces described as a gray level in a three dimensional image.

The output mesh conforms to some size and shape criteria which are customizable. The criteria are passed to the mesher through a parameter whose type is a model of the concept *SurfaceMeshFacetsCriteria_3*.

The meshing algorithm is a Delaunay refinement process which is mainly guided by the criteria. The output mesh may offer some guarantees, as being manifold, homeomorphic to the surface or within a given Hausdorff distance. However, these guarantees depend on the quality of the input surface (smoothness, with or without boundary, manifold or not), the type and values of the given criteria. The behavior of the refinement process can also be influenced through a tag, which allows for instance to enforce the manifold property of the output mesh while avoiding an over-refinement of the mesh.

48.8 Classified Reference Pages

Concepts

Classes

Tag Classes

Function Templates

48.9 Alphabetical List of Reference Pages

Class

CGAL::Gray_level_image_3<FT, Point>

Definition

A 3D gray image is a tri-dimensional array that associates a scalar value to each triple of integer (x, y, z) in the range of the image. A trilinear interpolation algorithm provides a map $f : \mathbb{R}^3 \longrightarrow \mathbb{R}$.

The class *Gray_level_image_3<FT, Point>* is a 3D gray image loader and a model of the concept *ImplicitFunction*. An object of the class *Gray_level_image_3<FT, Point>* is created with a parameter *iso* and then its *operator()* implements the function *sign of (f(p) - iso)*, for $p \in \mathbb{R}^3$. Plugging such a function in the creation of the *Implicit_surface_3* object given as parameter to *make_surface_mesh* yields a mesh approximating the level with value *iso* in the input 3D gray image.

Gray_level_image_3<FT, Point> provides an interface with an auxiliary library called *CGALimageIO*. An executable that uses *Gray_level_image_3<FT, Point>* must be linked with the *CGALimageIO* library. This library is shipped with CGAL and will be linked automatically by any makefile created by the script *cgal_create_makefile*.

The library *CGALimageIO* and therefore *Gray_level_image_3<FT, Point>* support several types of 3D images: INRIMAGE (extension .inr[.gz]), GIS (extension .dim, of .ima[.gz]), and ANALYZE (extension .hdr, or .img[.gz]).

#include <CGAL/Gray_level_image_3.h>

Is Model for the Concepts

ImplicitFunction

Types

Gray_level_image_3<FT, Point>:: FT	the numerical type *FT*
Gray_level_image_3<FT, Point>:: FT	the point type *Point*

Creation

Gray_level_image_3<FT, Point> image(const char filename, FT iso_value);*

filename is the path to a file of a type supported by *ImageIO*.
iso_value is an isovalue of f.

See Also

ImplicitFunction,
Implicit_surface_3<Traits, Function>,
make_surface_mesh

ImplicitFunction

Concept

Definition

The concept ImplicitFunction describes a function object whose *operator()* computes the values of a function $f : \mathbb{R}^3 \longrightarrow \mathbb{R}$.

Types

ImplicitFunction:: FT	Number type
ImplicitFunction:: Point	Point type

Operations

FT	*function(Point p)*	Returns the value $f(p)$, where $p \in \mathbb{R}^3$.

Has Models

Gray_level_image_function,
any pointer to a function of type *FT (*)(Point)*.

See Also

Implicit_surface_3<Traits, Function>,
make_surface_mesh

Class

CGAL::Implicit_surface_3<Traits, Function>

Definition

The class *Implicit_surface_3<Traits, Function>* implements a surface described as the zero level set of a function $f : \mathbb{R}^3 \longrightarrow \mathbb{R}$.

For this type of surface, the library provides a partial specialization of the surface mesher traits generator: *Surface_mesh_traits_generator_3<Implicit_surface_3<Traits, Function> >*, that provides a traits class, model of the concept *SurfaceMeshTraits_3*, to be used by the surface mesher.

The parameter *Traits* is a traits class that has to be implemented with a model of *ImplicitSurfaceTraits_3*. Actually, this traits class implements the oracle needed by the surface mesher: the types, predicates and constructors provided in *Traits* are passed by the surface mesher traits generator to the generated the traits class used by the surface mesh generator.

The template parameter *Function* stands for a model of the concept *ImplicitFunction*. The number type *Function::FT* has to match the type *Traits::FT*.

#include <CGAL/Implicit_surface_3.h>

Creation

Implicit_surface_3<Traits, Function> surface(Function f, Sphere_3 bounding_sphere, FT error_bound = FT(1e-3))

f is the object of type *Function* that represents the implicit surface.
bounding_sphere is a bounding sphere of the implicit surface. The evaluation of f at the center c of this sphere must be negative: $f(c) < 0$.
error_bound is a relative error bound used to compute intersection points between the implicit surface and query segments. This bound is used in the default generated traits class. In this traits class, the intersection points between the surface and segments/rays/line are constructed by dichotomy. The dichotomy is stopped when the size of the intersected segment is less than the product of *error_bound* by the radius of *bounding_sphere*.

See Also

make_surface_mesh,
Surface_3
Surface_mesh_traits_generator_3<Surface>,
ImplicitSurfaceTraits,
ImplicitFunction.

ImplicitSurfaceTraits_3

Definition

The concept ImplicitSurfaceTraits_3 describes the requirements of the traits class to be plugged as *Traits* in *Implicit_surface_3<Traits, Function>*.

When *make_surface_mesh* is called with a surface of type *Implicit_surface_3<Traits,Function>*, the surface mesher traits generator generates automatically a traits class that is a model of *SurfaceMeshTraits_3*. Actually, the concept ImplicitSurfaceTraits_3 provides the types, predicates and constructors that are passed to the generated model of *SurfaceMeshTraits_3*.

Types

ImplicitSurfaceTraits_3:: FT	The numerical type. It must be model of *SqrtFieldNumberType* and constructible from a *double*.
ImplicitSurfaceTraits_3:: Point_3	The point type. This point type must have a constructor *Point_3(FT, FT, FT)*.
ImplicitSurfaceTraits_3:: Line_3	The line type.
ImplicitSurfaceTraits_3:: Ray_3	The ray type.
ImplicitSurfaceTraits_3:: Segment_3	The segment type.
ImplicitSurfaceTraits_3:: Vector_3	The vector type.
ImplicitSurfaceTraits_3:: Sphere_3	The sphere type.

ImplicitSurfaceTraits_3:: Compute_scalar_product_3

A function object that provides the operator
FT operator()(Vector_3 v, Vector_3 w) which returns the scalar (inner) product of the two vectors *v* and *w*.

ImplicitSurfaceTraits_3:: Compute_squared_distance_3

A function object that provides the operator
FT operator()(Point_3, Point_3) which returns the squared distance between two points.

ImplicitSurfaceTraits_3:: Compute_squared_radius_3

A function object providing the operator
FT operator()(const Sphere_3& s) which returns the squared radius of *s*.

ImplicitSurfaceTraits_3:: Construct_center_3

A function object providing the operator
Point_3 operator()(const Sphere_3& s) which computes the center of the sphere *s*.

ImplicitSurfaceTraits_3:: Construct_midpoint_3

A function object providing the operator
Point_3 operator()(const Point_3& p, const Point_3& q) which computes the midpoint of the segment *pq*.

ImplicitSurfaceTraits_3:: Construct_point_on_3

A function object providing the following operators:
Point_3 operator()(const Line_3& l,int i); which returns an arbitrary point on *l*. It holds *point(i) == point(j)*, iff *i==j*. Furthermore, is directed from *point(i)* to *point(j)*, for all $i < j$.
Point_3 operator()(const Ray_3& r,int i); which returns a point on *r*. *point(0)* is the source, *point(i)*, with $i > 0$, is different from the source.
Precondition: $i \geq 0$.
Point_3 operator()(const Segment_3& s,int i); which returns source or target of *s*: *point(0)* returns the source of *s*, *point(1)* returns the target of *s*. The parameter *i* is taken modulo 2, which gives easy access to the other end point.

ImplicitSurfaceTraits_3:: Construct_segment_3

A function object providing the operators
Segment_3 operator()(const Point_3 &p, const Point_3 &q); which returns a segment with source *p* and target *q*. It is directed from the source towards the target.

ImplicitSurfaceTraits_3:: Construct_scaled_vector_3

A function object providing the operator
Vector_3 operator()(const Vector_3 &v, const FT& scale) which returns the vector *v* scaled by a factor *scale*.

ImplicitSurfaceTraits_3:: Construct_translated_point_3

A function object providing the operator
Point_3 operator()(const Point_3& p, const Vector_3& v) which returns the point obtained by translating *p* by the vector *v*.

ImplicitSurfaceTraits_3:: Construct_vector_3

A function object providing the operator
Vector_3 operator()(const Point_3 &a, const Point_3 &b) which returns the vector *b-a*.

ImplicitSurfaceTraits_3:: Has_on_bounded_side_3

A function object providing the operator
bool operator()(const Sphere_3&s, const Point_3&p); which returns true iff *p* lies on the bounded side of *s*.

Operations

The following functions give access to the predicate and construction objects:

Compute_scalar_product_3	*traits.compute_scalar_product_3_object()*
Compute_squared_distance_3	*traits.compute_squared_distance_3_object()*
Compute_squared_radius_3	*traits.compute_squared_radius_3_object()*
Construct_center_3	*traits.construct_center_3_object()*
Construct_midpoint_3	*traits.construct_midpoint_3_object()*
Construct_point_on_3	*traits.construct_point_on_3_object()*
Construct_scaled_vector_3	*traits.construct_scaled_vector_3_object()*
Construct_segment_3	*traits.construct_segment_3_object()*
Construct_translated_point_3	*traits.construct_translated_point_3_object()*
Construct_vector_3	*traits.construct_vector_3_object()*
Has_on_bounded_side_3	*traits.has_on_bounded_side_3_object()*

Has Models

Any CGAL Kernel.

See Also

Implicit_surface_3<Traits, Function>,
make_surface_mesh

CGAL::make_surface_mesh

#include <CGAL/make_surface_mesh.h>

Definition

The function *make_surface_mesh* is a surface mesh generator, that is a function to build a two dimensional mesh approximating a surface.

The library provides two overloaded version of this function:

template <class SurfaceMeshC2T3, class Surface, class FacetsCriteria, class Tag >
void make_surface_mesh(SurfaceMeshC2T3& c2t3,
Surface surface,
FacetsCriteria criteria,
Tag,
int initial_number_of_points = 20)

template <class SurfaceMeshC2T3, class SurfaceMeshTraits, class FacetsCriteria, class Tag >
void make_surface_mesh(SurfaceMeshC2T3& c2t3,
SurfaceMeshTraits::Surface_3 surface,
SurfaceMeshTraits traits,
FacetsCriteria criteria,
Tag,
int initial_number_of_points = 20)

Parameters

The template parameter *SurfaceMeshC2T3* is required to be a model of the concept *SurfaceMeshComplex_2InTriangulation_3*, a data structure able to represent a two dimensional complex embedded in a three dimensional triangulation. The argument *c2t3* of type *SurfaceMeshC2T3*, passed by reference to the surface mesh generator, is used to maintain the current approximating mesh and it stores the final mesh at the end of the procedure. The type *SurfaceMeshC2T3* is in particular required to provide a type *SurfaceMeshC2T3::Triangulation_3* for the three dimensional triangulation embedding the surface mesh. The vertex and cell base classes of the triangulation *SurfaceMeshC2T3::Triangulation_3* are required to be models of the concepts *SurfaceMeshVertexBase_3* and *SurfaceMeshCellBase_3* respectively.

The template parameter *Surface* stands for the surface type. This type has to be a model of the concept *Surface_3*.

The knowledge on the surface, required by the surface mesh generator is encapsulated in a traits class. Actually, the mesh generator accesses the surface to be meshed through this traits class only. The traits class is required to be a model of the concept *SurfaceMeshTraits_3*.

In the first version of *make_surface_mesh* the surface type is a template parameter *Surface* and the surface mesh generator traits type is automatically generated form the surface type through the class *Surface_mesh_traits_generator_3<Surface>*.

The difference between the two overloaded versions of *make_surface_mesh* can be explained as follows

- In the first overloaded version of of *make_surface_mesh*, the surface type is given as template parameter (*Surface*) and the *surface* to be meshed is passed as parameter to the mesh generator. In that case the surface mesh generator traits type is automatically generated form the surface type by an auxiliary class called the *Surface_mesh_traits_generator_3*.
- In the second overloaded version of *make_surface_mesh*, the surface mesh generator traits type is provided by the template parameter *SurfaceMeshTraits_3* and the surface type is obtained from this traits type. Both the surface and the traits are passed to the mesh generator as arguments.

The first overloaded version can be used whenever the surface type either provides a nested type *Surface::Surface_mesher_traits_3* that is a model of *SurfaceMeshTraits_3* or is a surface type for which a specialization of the traits generator *Surface_mesh_traits_generator_3*<*Surface*> is provided. Currently, the library provides partial specializations of *Surface_mesher_traits_generator_3*<*Surface*> for implicit surfaces (*Implicit_surface_3*<*Traits, Function*>) and gray level images (*Gray_level_image_3*<*FT, Point*>).

The template parameter *FacetsCriteria* has to be a model of the concept *SurfaceMeshFacetsCriteria_3*. The argument of type *FacetsCriteria* passed to the surface mesh generator specifies the size and shape requirements on the output surface mesh.

The template parameter *Tag* is a tag whose type affects the behavior of the meshing algorithm. The function *make_surface_mesh* has specialized versions for the following tag types:
- *Manifold_tag*: the output mesh is guaranteed to be a manifold surface without boundary.
- *Manifold_with_boundary_tag*: the output mesh is guaranteed to be manifold but may have boundaries.
- *Non_manifold_tag*: the algorithm relies on the given criteria and guarantees nothing else.

The Delaunay refinement process is started with an initial set of points which is the union of two sets: the set of vertices in the initial triangulation pointed to by the *c2t3* argument and a set of points provided by the traits class. The optional parameter *initial_number_of_points* allows to monitor the number of points in this second set. (This parameter is passed to the *operator()* of the constructor object *Construct_initial_points* in the traits class.) The meshing algorithm requires that the initial set of points includes at least one point on each connected components of the surface to be meshed. one.

See Also

SurfaceMeshComplex_2InTriangulation_3
SurfaceMeshCellBase_3
SurfaceMeshVertexBase_3
Surface_3
SurfaceMeshFacetsCriteria_3
Surface_mesh_default_triangulation_3

Class

CGAL::Manifold_tag

Definition

The class *Manifold_tag* is a tag class used to monitor the surface meshing algorithm. When instantiated with the tag *Manifold_tag* the function template *make_surface_mesh* ensures that the output mesh is a manifold surface without boundary.

#include <CGAL/make_surface_mesh.h>

See Also

make_surface_mesh
Manifold_with_boundary_tag
Non_manifold_tag

CGAL::Manifold_with_boundary_tag

Definition

The class *Manifold_with_boundary_tag* is a tag class used to monitor the surface meshing algorithm. When instantiated with the tag *Manifold_with_boundary_tag*, the function template *make_surface_mesh* ensures that the output mesh is a manifold surface but it may have boundaries.

#include <CGAL/make_surface_mesh.h>

See Also

make_surface_mesh
Manifold_tag
Non_manifold_tag

Class

CGAL::Non_manifold_tag

Definition

The class *Non_manifold_tag* is a tag class used to monitor the surface meshing algorithm. When instantiated with the tag *Non_manifold_tag* the function template *make_surface_mesh* does not ensure that the output mesh is a manifold surface. The manifold property of output mesh may nevertheless result from the choice of appropriate meshing criteria.

#include <CGAL/make_surface_mesh.h>

See Also

make_surface_mesh
Manifold_tag
Manifold_with_boundary_tag

Surface_3

Concept

Definition

The concept Surface_3 describes the types of surfaces to be meshed. The surface types are required to be copy constructible and assignable.

Types

In addition, surface types are required

- either to provide a nested type: *Surface_3:: Surface_mesher_traits_3* a model of *SurfaceMesherTraits_3*
- or to be a surface type for which a specialization of the traits generator *Surface_mesh_traits_generator_3<Surface>* exists.

Has Models

Implicit_surface_3<Traits, Function>

See Also

make_surface_mesh,
SurfaceMeshTraits_3
Surface_mesh_traits_generator_3<Surface>

Class

CGAL::Surface_mesh_cell_base_3<Gt,Cb>

Definition

The class *Surface_mesh_cell_base_3<Gt,Cb>* is a model of the concept *SurfaceMeshCellBase_3*. It is designed to serve as vertex base class in a triangulation class *Tr* plugged in a *Surface_mesh_complex_2_in_triangulation_3<Tr>* class.

The first template parameter is the geometric traits class.

The second template parameter is a base class. It has to be a model of the concept *TriangulationCellBase_3* and defaults to *Triangulation_cell_base_3 <GT>*.

#include <CGAL/Surface_mesh_cell_base_3.h>

Is Model for the Concepts

SurfaceMeshCellBase_3

Inherits From

Cb

See Also

SurfaceMeshComplex_2InTriangulation_3
Surface_mesh_complex_2_in_triangulation_3<Tr>
SurfaceMeshTriangulation_3
make_surface_mesh

SurfaceMeshCellBase_3

Concept

Definition

The concept SurfaceMeshCellBase_3 describes the cell base type of the three dimensional triangulation used to embed the surface mesh.

More precisely, the first template parameter *SurfaceMeshC2T3* of the surface mesher *make_surface_mesh* is a model of the concept *SurfaceMeshComplex_2InTriangulation_3* which describes a data structure to store a pure two dimensional complex embedded in a three dimensional triangulation. In particular, the type *SurfaceMeshC2T3* is required to provide a three dimensional triangulation type *SurfaceMeshC2T3::Triangulation_3*. The concept SurfaceMeshCellBase_3 describes the cell base type required in this triangulation type.

Refines

TriangulationCellBase_3

The concept SurfaceMeshCellBase_3 adds four markers to mark the facets of the triangulation that belong to the two dimensional complex, and four markers that are helpers used in some operations to mark for instance the facets that have been visited.

This concept also provides storage for the center of a Delaunay surface ball. Given a surface and a 3D Delaunay triangulation, a Delaunay surface ball is a ball circumscribed to a facet of the triangulation and centered on the surface and empty of triangulation vertices. Such a ball does exist when the facet is part of the restriction to the surface of a three dimensional triangulation. In the following we call *surface center* of a facet, the center of its biggest Delaunay surface ball.

Types

SurfaceMeshCellBase_3:: Point	The point type, required to match the point type of the three dimensional triangulation in which the surface mesh is embedded.

Creation

Operations

bool *cell.is_facet_on_surface(int i)*

returns *true*, if *facet(i)* is in the 2D complex.

void *cell.set_facet_on_surface(int i, bool b)*

Sets *facet(i)* as part of the 2D complex, if *b* is *true*, and *NOT_IN_COMPLEX*, otherwise.

bool *cell.is_facet_visited(int i)*

Returns *true*, if *facet(i)* has been visited, *false* otherwise.

void	*cell.set_facet_visited(int i, bool b)*	
		Marks *facet(i)* as visited, if *b* is *true*, and non visited otherwise.
Point	*cell.get_facet_surface_center(int i)*	
		Returns a const reference to the surface center of *facet(i)*.
void	*cell.set_facet_surface_center(int i, Point p)*	
		Sets point *p* as the surface center of *facet(i)*.

Has Models

Surface_mesh_cell_base_3<Gt,Vb>
Surface_mesh_default_triangulation_3::Cell

See Also

SurfaceMeshTriangulation_3
SurfaceMeshComplex_2InTriangulation_3
Surface_mesh_complex_2_in_triangulation_3<Tr>
Surface_mesh_default_triangulation_3
make_surface_mesh

CGAL::Surface_mesh_complex_2_in_triangulation_3<Tr, Edge_info = void>

Class

Definition

The class *Surface_mesh_complex_2_in_triangulation_3<Tr, Edge_info = void>* implements a data structure to store the restricted Delaunay triangulation used by the surface mesh generator. The restricted Delaunay triangulation is stored as a two dimensional complex embedded in a three dimensional triangulation.

The class *Surface_mesh_complex_2_in_triangulation_3<Tr, Edge_info = void>* is a model of the concept *SurfaceMeshComplex_2InTriangulation_3* and can be plugged as the template parameter *C2T3* in the function template *make_surface_mesh*.

The template parameter *Tr* has to be instantiated with a model of the concept *SurfaceMeshTriangulation_3*. (Any three dimensional triangulation of CGAL is a model of *Triangulation_3* provided that its vertex and cell base class be models of the concept *SurfaceMeshVertexBase_3* and *SurfaceMeshCellBase_3* respectively.)

#include <CGAL/Surface_mesh_complex_2_in_triangulation_3.h>

Is Model for the Concepts

SurfaceMeshComplex_2InTriangulation_3

See Also

make_surface_mesh
SurfaceMeshTriangulation_3

SurfaceMeshComplex_2InTriangulation_3

Definition

The concept SurfaceMeshComplex_2InTriangulation_3 describes a data structure designed to represent a two dimensional pure complex embedded in a three dimensional triangulation.

A *complex* is a set C of faces such that:
- any subface of a face in C is a face of C
- two faces of C are disjoint or share a common subface
The complex is *two dimensional*, if its faces have dimension at most two. It is *pure* if any face in the complex is a subface of some face of maximal dimension. Thus, a two dimensional pure complex is a set of facets together with their edges and vertices. A two dimensional pure complex embedded in a three dimensional triangulation is a subset of the facets of this triangulation, together with their edges and vertices.

The concept SurfaceMeshComplex_2InTriangulation_3 is particularly suited to handle surface meshes obtained as the restriction to a surface of a three dimensional Delaunay triangulation. A model of this concept is a type to be plugged as first template parameter in the function template *make_surface_mesh*.

Types

SurfaceMeshComplex_2InTriangulation_3 provides the following types.

SurfaceMeshComplex_2InTriangulation_3:: Triangulation

The type of the embedding 3D triangulation. Must be a model of *SurfaceMeshTriangulation_3*.

typedef Triangulation::Vertex_handle

Vertex_handle; The type of the embedding triangulation vertex handles.

typedef Triangulation::Cell_handle

Cell_handle; The type of the embedding triangulation cell handles.

typedef Triangulation::Facet

Facet; The type of the embedding triangulation facets.

typedef Triangulation::Edge

Edge; The type of the embedding triangulation edges.

typedef Triangulation::size_type

size_type; Size type (an unsigned integral type) *enum Face_status* { *NOT_IN_COMPLEX, BOUNDARY, REGULAR, SINGULAR*}*;*

A type to describe the status of a face (facet, edge, or vertex) with respect to the 2D pure complex. A *NOT_IN_COMPLEX* face does not belong to the 2D complex. Facets can only be *NOT_IN_COMPLEX* or *REGULAR* depending on whether they belong to the 2D complex on not. Edges and vertices can be *NOT_IN_COMPLEX*, *BOUNDARY*, *REGULAR* or *SINGULAR*. An edge in the complex is *BOUNDARY*, *REGULAR*, or *SINGULAR*, if it is incident to respectively 1, 2, or 3 or more facets in the complex. The status of a vertex is determined by the adjacency graph of the facets of the 2D complex incident to that vertex. The vertex of the 2D complex is *BOUNDARY*, if this adjacency graph is a simple path, it is *REGULAR*, if the adjacency graph is cyclic, and *SINGULAR* in any other case.

SurfaceMeshComplex_2InTriangulation_3:: Facet_iterator

An iterator type to visit the facets of the 2D complex.

SurfaceMeshComplex_2InTriangulation_3:: Edge_iterator

An iterator type to visit the edges of the 2D complex.

SurfaceMeshComplex_2InTriangulation_3:: Vertex_iterator

An iterator type to visit vertices of the 2D complex.

SurfaceMeshComplex_2InTriangulation_3:: Boundary_edges_iterator

An iterator type to visit the boundary edges of the 2D complex.

Creation

SurfaceMeshComplex_2InTriangulation_3 c2t3(Triangulation& t3);

Builds an empty 2D complex embedded in the triangulation *t3*

template < class FacetSelector>
SurfaceMeshComplex_2InTriangulation_3 c2t3(Triangulation& t3, FacetSelector select);

Builds a 2D complex embedded in the triangulation *t3*, including in the 2D complex the facets of *t3* for which the predicate *select* returns *true*.
The type *FacetSelector* must be a function object with an operator to select facets: *bool operator()(Facet f);*.

Member access

Triangulation&	*c2t3.triangulation()*	Returns the reference to the triangulation.

Modifications

void *c2t3.add_to_complex(Facet f)*

Adds facet *f* to the 2D complex.

void *c2t3.add_to_complex(Cell_handle c, int i)*

Adds facet *(c,i)* to the 2D complex.

void *c2t3.remove_from_complex(Facet f)*

Removes facet *f* from the 2D complex.

void *c2t3.remove_from_complex(Cell_handle c, int i)*

Removes facet *(c,i)* from the 2D complex.

Queries

Queries on the status of individual face with respect to the 2D complex.

size_type *c2t3.number_of_facets() const*

Returns the number of facets that belong to the 2D complex.

Face_status *c2t3.face_status(Facet f)*

Returns the status of the facet *f* with respect to the 2D complex.

Face_status *c2t3.face_status(Cell_handle c, int i)*

Returns the status of the facet *(c,i)* with respect to the 2D complex.

Face_status *c2t3.face_status(Edge e)*

Returns the status of edge *e* in the 2D complex.

Face_status *c2t3.face_status(Cell_handle c, int i, int j)*

Returns the status of edge *(c,i,j)* in the 2D complex.

Face_status *c2t3.face_status(Vertex_handle v)*

Returns the status of vertex *v* in the 2D complex.

bool *c2t3.is_in_complex(Facet f)*

Returns *true*, if the facet *f* belongs to the 2D complex.

bool	*c2t3.is_in_complex(Cell_handle c, int i)*	
		Returns *true*, if the facet *(c,i)* belongs to the 2D complex.
bool	*c2t3.is_in_complex(Edge e)*	
		Returns *true*, if the edge *e* belongs to the 2D complex.
bool	*c2t3.is_in_complex(Cell_handle c, int i, int j)*	
		Returns *true*, if the edge *(c,i,j)* belongs to the 2D complex.
bool	*c2t3.is_in_complex(Vertex_handle v)*	
		Returns *true*, if the vertex *v* belongs to the 2D complex.
bool	*c2t3.is_regular_or_boundary_for_vertices(Vertex_handle v)*	
		Returns true if the status of vertex *v* is *REGULAR* or *BOUNDARY*. *Precondition*: All the edges of the complex incident to *v* are *REGULAR* or *BOUNDARY*.

Traversal of the complex

The data structure provides iterators to visit the facets, edges and vertices of the complex. All those iterators are bidirectional and non mutable.

Facet_iterator	*c2t3.facets_begin()*	Returns an iterator with value type *Facet* to visit the facets of the 2D complex.
Facet_iterator	*c2t3.facets_end()*	Returns the past the end iterator for the above iterator.
Edge_iterator	*c2t3.edges_begin()*	Returns an iterator with value type *Edge* to visit the edges of the 2D complex which are not isolated.
Edge_iterator	*c2t3.edges_end()*	Returns the past the end iterator for the above iterator.
Boundary_edges_iterator		
	c2t3.boundary_edges_begin()	
		Returns an iterator with value type *Edge* to visit the boundary edges of the complex.
Boundary_edges_iterator		
	c2t3.boundary_edges_end()	
		Returns the past the end iterator for the above iterator.
Vertex_iterator	*c2t3.vertices_begin()*	Returns an iterator with value type *Vertex_handle* to visit the vertices of the 2D complex.
Vertex_iterator	*c2t3.vertices_end()*	Returns the past the end iterator for the above iterator.

template <class OutputIterator>

OutputIterator *c2t3.incident_facets(Vertex_handle v, OutputIterator facets)*

Copies the *Facet*s of the complex incident to v to the output iterator *facets*. Returns the resulting output iterator.
Precondition: *c2t3.triangulation().dimension()* $= 3$, $v \neq$ *Vertex_handle()*, *c2t3.triangulation().is_vertex(v)*.

The following function is the basic function to walk on the 2D complex

Facet *c2t3.neighbor(Facet f, int j)*

Returns the facet of the complex which is the neighbor of the facet f opposite to the vertex with index j of f. The vertices of the facet f = *(cell c, i)* are numbered (0,1,2) (according to the *vertex_triple_index(i,j)* member function of *Triangulation_3*) in such a way that facet f is oriented by the outward normal of tetraedra c. If there is no such neighbor, or if the edge is singular the functions returns *Facet()*.

Facet *c2t3.neighbor(Cell_handle c, int i, int j)*

Returns the facet of the complex which is the neighbor of the facet f opposite to the vertex with index j of f. See above.

Has Models

Surface_mesh_complex_2_in_triangulation_3<Tr>

See Also

make_surface_mesh.

SurfaceMeshFacetsCriteria_3

Concept

Definition

The Delaunay refinement process involved in the function template *make_surface_mesh* is guided by a set of refinement criteria. The concept SurfaceMeshFacetsCriteria_3 describes the type which handles those criteria. It corresponds to the requirements for the template parameter *FacetsCriteria* of the surface mesher function *make_surface_mesh<SurfaceMeshC2T3,Surface,FacetsCriteria,Tag>* .

Typically the meshing criteria are a set of elementary criteria, each of which has to be met by the facets of the final mesh. The meshing algorithm eliminates in turn *bad* facets, i.e., facets that do not meet all the criteria.

The size and quality of the final mesh depends on the order according to which bad facets are handled. Therefore, the meshing algorithm needs to be able to quantify the facet qualities and to compare the qualities of different faces. The concept SurfaceMeshFacetsCriteria_3 defines a type *Quality* designed to measure the quality of a mesh facet. Typically this quality is a multicomponent variable. Each component corresponds to one criterion and measures how much the facet deviates from meeting this criterion. Then, the comparison operator on qualities is just a lexicographical comparison. The meshing algorithm handles facets with lowest quality first. The qualities are computed by a function *is_bad(Facet f, Quality& q)*.

Types

SurfaceMeshFacetsCriteria_3:: Facet	The type of facets. This type has to match the *Facet* type in the triangulation type used by the mesher function. (This triangulation type is the type *SurfaceMeshC2T3::Triangulation* provided by the model of *SurfaceMeshComplex_2InTriangulation_3* plugged as first template parameter of *make_surface_mesh*).
SurfaceMeshFacetsCriteria_3:: Quality	Default constructible, copy constructible, assignable, and less-than comparable type.

Operations

bool *criteria.is_bad(Facet f, Quality& q)*

Assigns the quality of the facet *f* to *q*, and returns *true* is *f* does not meet the criteria.

Has Models

Surface_mesh_default_criteria_3<Tr>

See Also

make_surface_mesh<SurfaceMeshC2T3,Surface,FacetsCriteria,Tag>

Class

CGAL::Surface_mesh_default_criteria_3<Tr>

Definition

The class *Surface_mesh_default_criteria_3<Tr>* implements the most commonly used combination of meshing criteria. It involves mainly three criteria which are in order:

- a lower bound on the minimum angle in degrees of the surface mesh facets.
- an upper bound on the radius of surface Delaunay balls. A surface Delaunay ball is a ball circumscribing a facet, centered on the surface and empty of vertices. Such a ball exists for each facet of the current surface mesh. Indeed the current surface mesh is the Delaunay triangulation of the current sampling restricted to the surface which is just the set of facets in the three dimensional Delaunay triangulation of the sampling that have a Delaunay surface ball.
- an upper bound on the center-center distances of the surface mesh facets. The center-center distance of a surface mesh facet is the distance between the facet circumcenter and the center of its surface Delaunay ball.

#include <CGAL/Surface_mesh_default_criteria_3.h>

Is Model for the Concepts

SurfaceMeshFacetsCriteria_3

Types

typedef Tr::FT	*FT;*	The numerical type.

Creation

Surface_mesh_default_criteria_3<Tr> criteria(FT angle_bound, FT radius_bound, FT distance_bound);

Returns a *Surface_mesh_default_criteria_3<Tr>* with *angle_bound*, *radius_bound*, *distance_bound* as bounds for the minimum facet angle in degrees, the radius of the surface Delaunay balls and the center-center distances respectively.

See Also

make_surface_mesh

CGAL::Surface_mesh_default_triangulation_3

Definition

The class *Surface_mesh_default_triangulation_3* is a model of the concept *SurfaceMeshTriangulation_3*, whose vertex base and cell base classes are models of the concepts *SurfaceMeshVertexBase_3* and *SurfaceMeshCellBase_3* respectively.

#include <CGAL/Surface_mesh_default_triangulation_3.h>

Is Model for the Concepts

SurfaceMeshTriangulation_3

See Also

Surface_mesh_complex_2_in_triangulation_3<Tr>
make_surface_mesh

Class

Concept

SurfaceMeshTraits_3

Definition

The concept SurfaceMeshTraits_3 describes the knowledge that is required on the surface to be meshed. A model of this concept implements an oracle that is able to tell whether a segment (or a ray, or a line) intersects the surface or not and to compute some intersection points if any. The concept SurfaceMeshTraits_3 also includes a constructor able to provide a small set of initial points on the surface.

Types

SurfaceMeshTraits_3:: Point_3	The type of points. This type is required to match the point type of the three dimensional embedding triangulation *C2T3::Triangulation_3*.
SurfaceMeshTraits_3:: Segment_3	The type of segments.
SurfaceMeshTraits_3:: Ray_3	The type of rays.
SurfaceMeshTraits_3:: Line_3	The type of lines.
SurfaceMeshTraits_3:: Surface_3	The surface type.
SurfaceMeshTraits_3:: Intersect_3	A model of this type provides the operator *CGAL::Object operator()(Surface_3 surface, Type1 type1)* to compute the intersection of the surface with an object of type *Type1* which may be *Segment_3*, *Ray_3* or *Line_3* .

SurfaceMeshTraits_3:: Construct_initial_points

A model of this type provides the following operators to construct initial points on the surface:
template <class OutputIteratorPoints>
OutputIteratorPoints operator()(OutputIteratorPoints pts)
which outputs a set of points on the surface,
template <class OutputIteratorPoints>
OutputIteratorPoints operator() (OutputIteratorPoints pts, int n)
which outputs a set of *n* points on the surface.

Operations

The following functions give access to the construction objects:

Intersect_3	*traits.intersect_3_object()*
Construct_initial_points	*traits.construct_initial_points_object()*

Has Models

Surface_mesh_traits_generator_3<Surface>::Type

See Also

make_surface_mesh

Class

CGAL::Surface_mesh_traits_generator_3<Surface>

Definition

The class *Surface_mesh_traits_generator_3<Surface>* provides a type *Type*, that is a model of the concept *SurfaceMeshTraits_3* for the surface type *Surface*.

The type *Surface* is required to be a model of the concept *Surface_3*, which means that it is copy constructible and assignable. In addition, a *Surface* type is required

- either to provide a nested type *Surface::Surface_mesher_traits_3* that is a model of *SurfaceMeshTraits_3*
- or to be a surface type for which a specialization of the traits generator *Surface_mesh_traits_generator_3<Surface>* exists.

Currently, the library provides partial specializations of the traits generator for implicit surfaces (*Implicit_surface_3<Traits, Function>*) and gray level images (*Gray_level_image_3<FT, Point>*).

#include <CGAL/Surface_mesh_traits_generator_3.h>

Surface_mesh_traits_generator_3<Surface>:: Type

A model of the concept *SurfaceMeshTraits_3*.

See Also

SurfaceMeshTraits_3 make_surface_mesh

SurfaceMeshTriangulation_3

Concept

Definition

The concept SurfaceMeshTriangulation_3 describes the triangulation type used by the surface mesher *make_surface_mesh* to represent the three dimensional triangulation embedding the surface mesh. Thus, this concept describes the requirements for the triangulation type *SurfaceMeshC2T3::Triangulation* nested in the model of *SurfaceMeshComplex2InTriangulation3* plugged as the template parameter *SurfaceMeshC2T3* of *make_surface_mesh*. It also describes the requirements for the triangulation type plugged in the class *Surface_mesh_complex_2_in_triangulation_3<Tr>*.

Types

SurfaceMeshTriangulation_3:: Point — The point type. It must be DefaultConstructible, CopyConstructible and Assignable.

Vertices and *cells* of the triangulation are manipulated via handles, which support the two dereference operators *operator** and *operator->*.

SurfaceMeshTriangulation_3:: Vertex_handle — Handle to a data representing a *vertex*. *Vertex_handle* must be a model of *Handle* and its *value type* must be model of *TriangulationDataStructure_3::Vertex*.

SurfaceMeshTriangulation_3:: Cell_handle — Handle to a data representing a *cell*. *Cell_handle* must be a model of *Handle* and its *value type* must be model of *TriangulationDataStructure_3::Cell*.

typedef CGAL::Triple<Cell_handle, int, int>
Edge; — The edge type.

typedef std::pair<Cell_handle, int>
Facet; — The facet type.

The following iterators allow one to visit all finite vertices, edges and facets of the triangulation.

SurfaceMeshTriangulation_3:: Finite_vertices_iterator

Iterator over finite vertices

SurfaceMeshTriangulation_3:: Finite_edges_iterator

Iterator over finite edges

SurfaceMeshTriangulation_3:: Finite_facets_iterator

Iterator over finite facets

SurfaceMeshTriangulation_3:: Geom_traits — The geometric traits class. Must be a model of *DelaunayTriangulationTraits_3*.

Creation

SurfaceMeshTriangulation_3 t;	default constructor.
SurfaceMeshTriangulation_3 t(tr);	Copy constructor. All vertices and faces are duplicated.

Assignment

SurfaceMeshTriangulation_3 &

	$t = tr$	The triangulation *tr* is duplicated, and modifying the copy after the duplication does not modify the original. The previous triangulation held by *t* is deleted.
void	*t.clear()*	Deletes all finite vertices and all cells of *t*.

Access Functions

int	*t.dimension() const*	Returns the dimension of the affine hull.

DelaunayTriangulationTraits_3

	t.geom_traits() const	Returns a const reference to a model of *DelaunayTriangulationTraits_3*.

Voronoi diagram

Object	*t.dual(Facet f) const*	Returns the dual of facet *f*, which is in dimension 3: either a segment, if the two cells incident to *f* are finite, or a ray, if one of them is infinite; in dimension 2: a point.

Queries

A point *p* is said to be in conflict with a cell *c* in dimension 3 (resp. a facet *f* in dimension 2) iff *t.side_of_sphere(c, p)* (resp. *t.side_of_circle(f, p)*) returns *ON_BOUNDED_SIDE*. The set of cells (resp. facets in dimension 2) which are in conflict with *p* is connected, and it forms a hole.

template <class OutputIteratorBoundaryFacets, class OutputIteratorCells, class OutputIteratorInternalFacets>

Triple<OutputIteratorBoundaryFacets, OutputIteratorCells, OutputIteratorInternalFacets>

t.find_conflicts(Point p,
Cell_handle c,
OutputIteratorBoundaryFacets bfit,
OutputIteratorCells cit,

OutputIteratorInternalFacets ifit)

Computes the conflict hole induced by p. The starting cell (resp. facet) c must be in conflict. Then this function returns respectively in the output iterators:
– *cit*: the cells (resp. facets) in conflict.
– *bfit*: the facets (resp. edges) on the boundary, that is, the facets (resp. edges) *(t, i)* where the cell (resp. facet) t is in conflict, but *t->neighbor(i)* is not.
– *ifit*: the facets (resp. edges) inside the hole, that is, delimiting two cells (resp facets) in conflict.
Returns the *Triple* composed of the resulting output iterators.

The following iterators allow the user to visit facets, edges and vertices of the triangulation.

Finite_vertices_iterator

t.finite_vertices_begin() const

Starts at an arbitrary finite vertex. Then ++ and -- will iterate over finite vertices. Returns *finite_vertices_end()* when *t.number_of_vertices()* $= 0$.

Finite_vertices_iterator

t.finite_vertices_end() const

Past-the-end iterator

Finite_edges_iterator *t.finite_edges_begin() const*

Starts at an arbitrary finite edge. Then ++ and -- will iterate over finite edges. Returns *finite_edges_end()* when *t.dimension()* < 1.

Finite_edges_iterator *t.finite_edges_end() const*

Past-the-end iterator

Finite_facets_iterator

t.finite_facets_begin() const

Starts at an arbitrary finite facet. Then ++ and -- will iterate over finite facets. Returns *finite_facets_end()* when *t.dimension()* < 2.

Finite_facets_iterator

t.finite_facets_end() const

Past-the-end iterator

template <class OutputIterator>

OutputIterator *t.incident_cells(Vertex_handle v, OutputIterator cells) const*

Copies the *Cell_handle*s of all cells incident to v to the output iterator *cells*. If *t.dimension()* < 3, then do nothing. Returns the resulting output iterator.
Precondition: $v \neq$ *Vertex_handle()*, *t.is_vertex(v)*.

template <class OutputIterator>
OutputIterator *t.incident_cells(Vertex_handle v, OutputIterator cells) const*

Copies the *Cell_handle*s of all cells incident to v to the output iterator *cells*. If *t.dimension()* < 3, then do nothing. Returns the resulting output iterator.

bool *t.is_vertex(Point p, Vertex_handle & v) const*

Tests whether p is a vertex of t by locating p in the triangulation. If p is found, the associated vertex v is given.

bool *t.is_edge(Vertex_handle u, Vertex_handle v, Cell_handle & c, int & i, int & j) const*

Tests whether *(u,v)* is an edge of t. If the edge is found, it gives a cell c having this edge and the indices i and j of the vertices u and v in c, in this order.
Precondition: u and v are vertices of t.

bool *t.is_infinite(const Vertex_handle v) const*

true, iff vertex v is the infinite vertex.

bool *t.is_infinite(const Cell_handle c) const*

true, iff c is incident to the infinite vertex.
Precondition: *t.dimension()* $= 3$.

Facet *t.mirror_facet(Facet f) const*

Returns the same facet viewed from the other adjacent cell.

int *t.vertex_triple_index(const int i, const int j)*

Return the indexes of the jth vertex of the facet of a cell opposite to vertex i.

Point location

Cell_handle *t.locate(Point query, Cell_handle start = Cell_handle())*

If the point *query* lies inside the convex hull of the points, the cell that contains the query in its interior is returned. If *query* lies on a facet, an edge or on a vertex, one of the cells having *query* on its boundary is returned.
If the point *query* lies outside the convex hull of the points, an infinite cell with vertices $\{p,q,r,\infty\}$ is returned such that the tetrahedron $(p,q,r,query)$ is positively oriented (the rest of the triangulation lies on the other side of facet (p,q,r)).
Note that locate works even in degenerate dimensions: in dimension 2 (resp. 1, 0) the *Cell_handle* returned is the one that represents the facet (resp. edge, vertex) containing the query point.
The optional argument *start* is used as a starting place for the search.

Cell_handle *t.locate(Point query,*
Locate_type & lt,
int & li,
int & lj,
Cell_handle start = Cell_handle())

If *query* lies inside the affine hull of the points, the *k*-face (finite or infinite) that contains *query* in its interior is returned, by means of the cell returned together with *lt*, which is set to the locate type of the query (*VERTEX, EDGE, FACET, CELL*, or *OUTSIDE_CONVEX_HULL* if the cell is infinite and *query* lies strictly in it) and two indices *li* and *lj* that specify the *k*-face of the cell containing *query*.
If the *k*-face is a cell, *li* and *lj* have no meaning; if it is a facet (resp. vertex), *li* gives the index of the facet (resp. vertex) and *lj* has no meaning; if it is and edge, *li* and *lj* give the indices of its vertices.
If the point *query* lies outside the affine hull of the points, which can happen in case of degenerate dimensions, *lt* is set to *OUTSIDE_AFFINE_HULL*, and the cell returned has no meaning. As a particular case, if there is no finite vertex yet in the triangulation, *lt* is set to *OUTSIDE_AFFINE_HULL* and *locate* returns the default constructed handle.
The optional argument *start* is used as a starting place for the search.

template <class CellIt>

Vertex_handle *t.insert_in_hole(Point p, CellIt cell_begin, CellIt cell_end, Cell_handle begin, int i)*

Creates a new vertex by starring a hole. It takes an iterator range [*cell_begin*; *cell_end*[of *Cell_handle*s which specifies a hole: a set of connected cells (resp. facets in dimension 2) which is star-shaped wrt p. (*begin*, i) is a facet (resp. an edge) on the boundary of the hole, that is, *begin* belongs to the set of cells (resp. facets) previously described, and *begin->neighbor(i)* does not. Then this function deletes all the cells (resp. facets) describing the hole, creates a new vertex v, and for each facet (resp. edge) on the boundary of the hole, creates a new cell (resp. facet) with v as vertex. Then *v->set_point(p)* is called and v is returned.

Precondition: *t.dimension()* ≥ 2, the set of cells (resp. facets in dimension 2) is connected, its boundary is connected, and p lies inside the hole, which is star-shaped wrt p.

Has Models

Any 3D Delaunay triangulation class of CGAL

See Also

Triangulation_3<TriangulationTraits_3,TriangulationDataStructure_3>
Delaunay_triangulation_3<DelaunayTriangulationTraits_3,TriangulationDataStructure_3>
SurfaceMeshComplex2InTriangulation3
Surface_mesh_complex_2_in_triangulation_3<Tr>
make_surface_mesh

CGAL::Surface_mesh_vertex_base_3<Gt,Vb>

Class

Definition

The class *Surface_mesh_vertex_base_3<Gt,Vb>* is a model of the concept *SurfaceMeshVertexBase_3*. It is designed to serve as vertex base class in a triangulation class *Tr* plugged in a *Surface_mesh_complex_2_in_triangulation_3<Tr>* class.

The first template parameter is the geometric traits class.

The second template parameter is a base class. It has to be a model of the concept *TriangulationVertexBase_3* and defaults to *Triangulation_vertex_base_3 <Gt>*.

#include <CGAL/Surface_mesh_vertex_base_3.h>

Is Model for the Concepts

SurfaceMeshVertexBase_3

Inherits From

Vb

See Also

SurfaceMeshComplex_2InTriangulation_3
Surface_mesh_complex_2_in_triangulation_3<Tr>
SurfaceMeshTriangulation_3
make_surface_mesh

SurfaceMeshVertexBase_3

Definition

The concept SurfaceMeshVertexBase_3 describes the vertex base type of the three dimensional triangulation used to embed the surface mesh.

More precisely, the first template parameter *SurfaceMeshC2T3* of the surface mesher *make_surface_mesh* is a model of the concept *SurfaceMeshComplex_2InTriangulation_3* which describes a data structure to store a pure two dimensional complex embedded in a three dimensional triangulation. In particular, the type *SurfaceMeshC2T3* is required to provide a three dimensional triangulation type *SurfaceMeshC2T3::Triangulation_3* The concept SurfaceMeshVertexBase_3 describes the vertex base type required in this triangulation type.

Refines

TriangulationVertexBase_3

The surface mesher algorithm issues frequent queries about the status of the vertices with respect to the two dimensional complex that represents the current surface approximation. The class SurfaceMeshVertexBase_3 offers a caching mechanism to answer more efficiently these queries. The caching mechanism includes two cached integers, which, when they are valid, store respectively the number of complex facets incident to the vertex and the number of connected components of the adjacency graph of those facets.

Creation

Operations

bool *vb.is_c2t3_cache_valid()*

Returns *true* if the cache is valid.

void *vb.invalidate_c2t3_cache()*

Invalidates the cache.

int *vb.cached_number_of_incident_facets()*

Returns the cached number of facets of the complex incident to the vertex.

int *vb.cached_number_of_components()*

This method concerns the adjacency graph of the facets of the complex incident to the vertex and returns a cached value for the number of connected components this graph.

Has Models

Surface_mesh_vertex_base_3<Gt,Vb>
Surface_mesh_default_triangulation_3::Vertex

See Also

SurfaceMesherComplex_2InTriangulation_3
Surface_mesh_complex_2_in_triangulation_3<Tr>
Surface_mesh_default_triangulation_3.

Function

CGAL::output_surface_facets_to_polyhedron

Definition

output_surface_facets_to_polyhedron() converts a surface reconstructed by *make_surface_mesh()* to a *Polyhedron_3<Traits>*.

#include <CGAL/IO/output_surface_facets_to_polyhedron.h>

template<class SurfaceMeshComplex_2InTriangulation_3, class Polyhedron>
bool *output_surface_facets_to_polyhedron(SurfaceMeshComplex_2InTriangulation_3 c2t3, Polyhedron& output_polyhedron)*

Gets reconstructed surface out of a *SurfaceMeshComplex_2InTriangulation_3* object.
This variant exports the surface as a polyhedron. It requires the surface to be manifold. For this purpose, you may call *make_surface_mesh*() with *Manifold_tag* or *Manifold_with_boundary_tag* parameter.
Template Parameters:
SurfaceMeshComplex_2InTriangulation_3: model of the *SurfaceMeshComplex_2InTriangulation_3* concept. *Polyhedron*: an instance of *Polyhedron_3<Traits>*.
Returns: true if the surface is manifold and orientable.
Parameters:
c2t3: Input surface. *output_polyhedron*: Output polyhedron.

See Also

CGAL::output_surface_facets_to_off . . . . page **??**

Chapter 49

Surface Reconstruction from Point Sets

Pierre Alliez, Laurent Saboret, Gaël Guennebaud

Contents

49.1 Introduction

This CGAL component implements a surface reconstruction method which takes as input point sets with oriented normals and computes an implicit function. We assume that the input points contain no outliers and little noise. The output surface mesh is generated by extracting an isosurface of this function with the CGAL Surface Mesh Generator [RY07] or potentially with any other surface contouring algorithm.

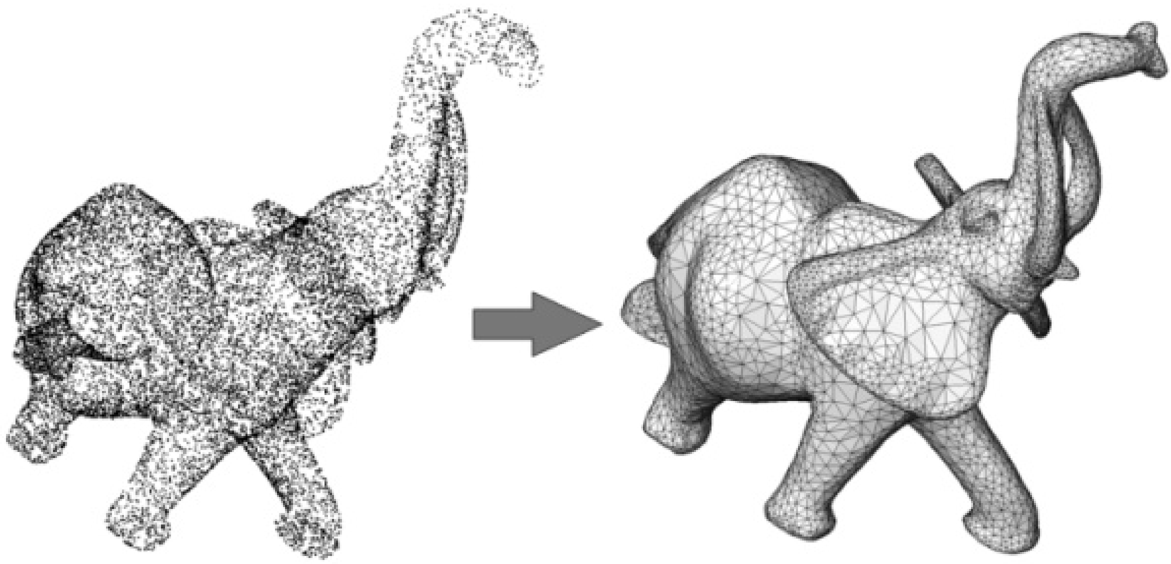


Figure 49.1: Poisson surface reconstruction. Left: 17K points sampled on the statue of an elephant with a Minolta laser scanner. Right: reconstructed surface mesh.

More specifically, the core surface reconstruction algorithm consists of computing an implicit function which is an approximate indicator function of the inferred solid (Poisson Surface Reconstruction - referred to as Poisson). Poisson is a two steps process: it requires solving for the implicit function before function evaluation.

49.2 Common Reconstruction Pipeline

Surface reconstruction from point sets is often a sequential process with the following steps: 1) Scanning and scan alignment produce a set of points or points with normals; 2) Outlier removal; 3) Simplification to reduce the number of input points; 4) Smoothing to reduce noise in the input data; 5) Normal estimation and orientation when the normals are not already provided by the acquisition device; and 6) Surface reconstruction.
CGAL provides algorithms for all steps listed above except alignment.
Chapter *Point_set_processing_3* 58 describes algorithms to pre-process the point set before reconstruction with functions devoted to the simplification, outlier removal, smoothing, normal estimation and normal orientation.

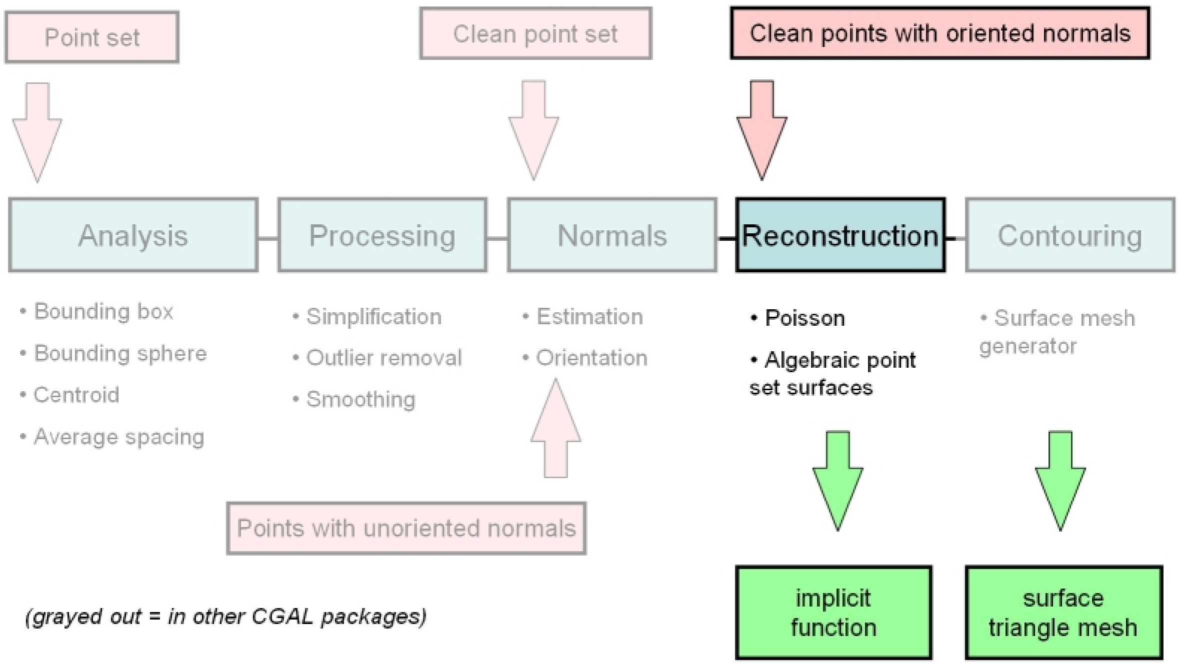


Figure 49.2: Common surface reconstruction pipeline.

49.3 Poisson

Given a set of 3D points with oriented normals (denoted oriented points in the sequel) sampled on the boundary of a 3D solid, the Poisson Surface Reconstruction method [KBH06] solves for an approximate indicator function of the inferred solid, whose gradient best matches the input normals. The output scalar function, represented in an adaptive octree, is then iso-contoured using an adaptive marching cubes.

CGAL implements a variant of this algorithm which solves for a piecewise linear function on a 3D Delaunay triangulation instead of an adaptive octree. The algorithm takes as input a set of 3D oriented points. It builds a 3D Delaunay triangulation from these points and refines it by Delaunay refinement so as to remove all badly shaped (non isotropic) tetrahedra and to tessellate a loose bounding box of the input oriented points. The normal of each Steiner point added during refinement is set to zero. It then solves for a scalar indicator function f represented as a piecewise linear function over the refined triangulation. More specifically, it solves for the Poisson equation $\Delta f = div(\mathbf{n})$ at each vertex of the triangulation using a sparse linear solver. Eventually, the CGAL surface mesh generator extracts an isosurface with function value set by default to be the median value of f at all input points.

49.3.1 Interface

The class template declaration is:

template<
class Gt>
class *Poisson_reconstruction_function*;

with
Gt: Geometric traits class.

Creation:

template<typename InputIterator, typename PointPMap, typename NormalPMap>
Poisson_reconstruction_function(InputIterator first, InputIterator beyond, PointPMap point_pmap, NormalPMap normal_pmap)

Creates a Poisson implicit function from the [first, beyond) range of points.
Template Parameters:
InputIterator: iterator over input points. *PointPMap*: is a model of *boost::ReadablePropertyMap* with a *value_type* = *Point_3*. It can be omitted if *InputIterator value_type* is convertible to *Point_3*. *NormalPMap*: is a model of *boost::ReadablePropertyMap* with a *value_type* = *Vector_3*.
Parameters:
first: iterator over the first input point. *beyond*: past-the-end iterator over the input points. *point_pmap*: property map to access the position of an input point. *normal_pmap*: property map to access the **oriented** normal of an input point.

The main operations are:

Sphere *bounding_sphere() const*

Returns a sphere bounding the inferred surface.

bool *compute_implicit_function()*

The function *compute_implicit_function*() must be called after each insertion of oriented points. It computes the piecewise linear scalar function operator() by: applying Delaunay refinement, solving for operator() at each vertex of the triangulation with a sparse linear solver, and shifting and orienting operator() such that it is 0 at all input points and negative inside the inferred surface.
Returns false if the linear solver fails.
The sparse solver used for this step is a parameter of the function. We recommend to use the class *CGAL::Eigen_solver_traits<T>* instantiated with the iterative conjugate gradient solver *Eigen::ConjugateGradient* for *double* (which is the default when Eigen (Page 28) is available and *CGAL_EIGEN3_ENABLED* is defined).

FT *value(Point p) const* Evaluates the implicit function at a given 3D query point.

Point *get_inner_point() const*

Returns an arbitrary point located inside the inferred surface.

See details in
CGAL::Poisson_reconstruction_function<GeomTraits>

49.3.2 Example

poisson_reconstruction_example.cpp reads a point set, creates a Poisson implicit function and reconstructs a surface.

```
#include <CGAL/trace.h>
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/Polyhedron_3.h>
#include <CGAL/IO/Polyhedron_iostream.h>
#include <CGAL/Surface_mesh_default_triangulation_3.h>
#include <CGAL/make_surface_mesh.h>
#include <CGAL/Implicit_surface_3.h>
#include <CGAL/IO/output_surface_facets_to_polyhedron.h>
#include <CGAL/Poisson_reconstruction_function.h>
#include <CGAL/Point_with_normal_3.h>
#include <CGAL/property_map.h>
#include <CGAL/IO/read_xyz_points.h>
#include <CGAL/compute_average_spacing.h>

#include <vector>
#include <fstream>

// Types
typedef CGAL::Exact_predicates_inexact_constructions_kernel Kernel;
typedef Kernel::FT FT;
typedef Kernel::Point_3 Point;
typedef CGAL::Point_with_normal_3<Kernel> Point_with_normal;
typedef Kernel::Sphere_3 Sphere;
typedef std::vector<Point_with_normal> PointList;
typedef CGAL::Polyhedron_3<Kernel> Polyhedron;
typedef CGAL::Poisson_reconstruction_function<Kernel>
   Poisson_reconstruction_function;
typedef CGAL::Surface_mesh_default_triangulation_3 STr;
typedef CGAL::Surface_mesh_complex_2_in_triangulation_3<STr> C2t3;
typedef CGAL::Implicit_surface_3<Kernel, Poisson_reconstruction_function>
    Surface_3;

int main(void)
{
    // Poisson options
    FT sm_angle = 20.0; // Min triangle angle in degrees.
    FT sm_radius = 30; // Max triangle size w.r.t. point set average
   spacing.
    FT sm_distance = 0.375; // Surface Approximation error w.r.t. point
   set average spacing.

    // Reads the point set file in points[].
    // Note: read_xyz_points_and_normals() requires an iterator over
   points
    // + property maps to access each point's position and normal.
```

```
 // The position property map can be omitted here as we use iterators
over Point_3 elements.
 PointList points;
 std::ifstream stream("data/kitten.xyz");
 if (!stream ||
     !CGAL::read_xyz_points_and_normals(
                              stream,
                              std::back_inserter(points),

CGAL::make_normal_of_point_with_normal_pmap(std::back_inserter(points))))
 {
   std::cerr << "Error: cannot read file data/kitten.xyz" <<
std::endl;
   return EXIT_FAILURE;
 }

 // Creates implicit function from the read points using the default
solver.
 // Note: this method requires an iterator over points
 // + property maps to access each point's position and normal.
 // The position property map can be omitted here as we use iterators
over Point_3 elements.
 Poisson_reconstruction_function function(
                              points.begin(), points.end(),

CGAL::make_normal_of_point_with_normal_pmap(points.begin()));

 // Computes the Poisson indicator function f()
 // at each vertex of the triangulation.
 if ( ! function.compute_implicit_function() )
   return EXIT_FAILURE;

 // Computes average spacing
 FT average_spacing = CGAL::compute_average_spacing(points.begin(),
points.end(),
                                                      6 /* knn = 1 ring
*/);

 // Gets one point inside the implicit surface
 // and computes implicit function bounding sphere radius.
 Point inner_point = function.get_inner_point();
 Sphere bsphere = function.bounding_sphere();
 FT radius = std::sqrt(bsphere.squared_radius());

 // Defines the implicit surface: requires defining a
 // conservative bounding sphere centered at inner point.
 FT sm_sphere_radius = 5.0 * radius;
 FT sm_dichotomy_error = sm_distance*average_spacing/1000.0; //
Dichotomy error must be << sm_distance
 Surface_3 surface(function,

Sphere(inner_point,sm_sphere_radius*sm_sphere_radius),
                      sm_dichotomy_error/sm_sphere_radius);
```

```
    // Defines surface mesh generation criteria
    CGAL::Surface_mesh_default_criteria_3<STr> criteria(sm_angle,  // Min
     triangle angle (degrees)

   sm_radius*average_spacing,  // Max triangle size

   sm_distance*average_spacing); // Approximation error

    // Generates surface mesh with manifold option
    STr tr; // 3D Delaunay triangulation for surface mesh generation
    C2t3 c2t3(tr); // 2D complex in 3D Delaunay triangulation
    CGAL::make_surface_mesh(c2t3,                                  //
   reconstructed mesh
                              surface,                              //
   implicit surface
                              criteria,                             //
   meshing criteria
                              CGAL::Manifold_with_boundary_tag());  //
   require manifold mesh

    if(tr.number_of_vertices() == 0)
      return EXIT_FAILURE;

    // saves reconstructed surface mesh
    std::ofstream out("kitten_poisson-20-30-0.375.off");
    Polyhedron output_mesh;
    CGAL::output_surface_facets_to_polyhedron(c2t3, output_mesh);
    out << output_mesh;

    return EXIT_SUCCESS;
}
```

File: examples/Surface_reconstruction_points_3/poisson_reconstruction_example.cpp

49.4 Contouring

The computed implicit functions can be iso-contoured to reconstruct a surface by using the CGAL surface mesh generator [RY07, BO05]:
CGAL::make_surface_mesh

The parameter *Tag* affects the behavior of *make_surface_mesh()*:
- *Manifold_tag*: the output mesh is guaranteed to be a manifold surface without boundary.
- *Manifold_with_boundary_tag*: the output mesh is guaranteed to be manifold and may have boundaries.
- *Non_manifold_tag*: the output mesh has no guarantee and hence is outputted as a polygon soup.

49.5 Output

The surface reconstructed by *make_surface_mesh()* is required to be a model of the concept *SurfaceMeshComplex_2InTriangulation_3*, a data structure devised to represent a two dimensional complex embedded into a three dimensional triangulation.

SurfaceMeshComplex_2InTriangulation_3 defines the methods to traverse the reconstructed surface, and e.g. convert it to a triangle soup.

Other CGAL components provide functions to write the reconstructed surface mesh to the Object File Format (OFF) [Phi96] and to convert it to a polyhedron (when it is manifold):
CGAL::output_surface_facets_to_off
CGAL::output_surface_facets_to_polyhedron

See *poisson_reconstruction_example.cpp* example above.

49.6 Case Studies

The surface reconstruction problem being inherently ill-posed, the proposed algorithm does not pretend to reconstruct all kinds of surfaces with arbitrary sampling conditions. This section provides the user with some hints about the ideal sampling and contouring conditions, and depicts some failure cases when these conditions are not matched.

49.6.1 Ideal Conditions

The user must keep in mind that the poisson surface reconstruction algorithm comprises two phases (computing the implicit function from the input point set and contouring an iso-surface of this function). Both require some care in terms of sampling conditions and parameter tuning.

Point Set

Ideally the current implementation of the Poisson surface reconstruction method expects a dense 3D oriented point set (typically matching the epsilon-sampling condition [BO05]) and sampled over a closed, smooth surface. Oriented herein means that all 3D points must come with consistently oriented normals to the inferred surface. Figures 49.3 and 49.4 illustrate cases where these ideal conditions are met.

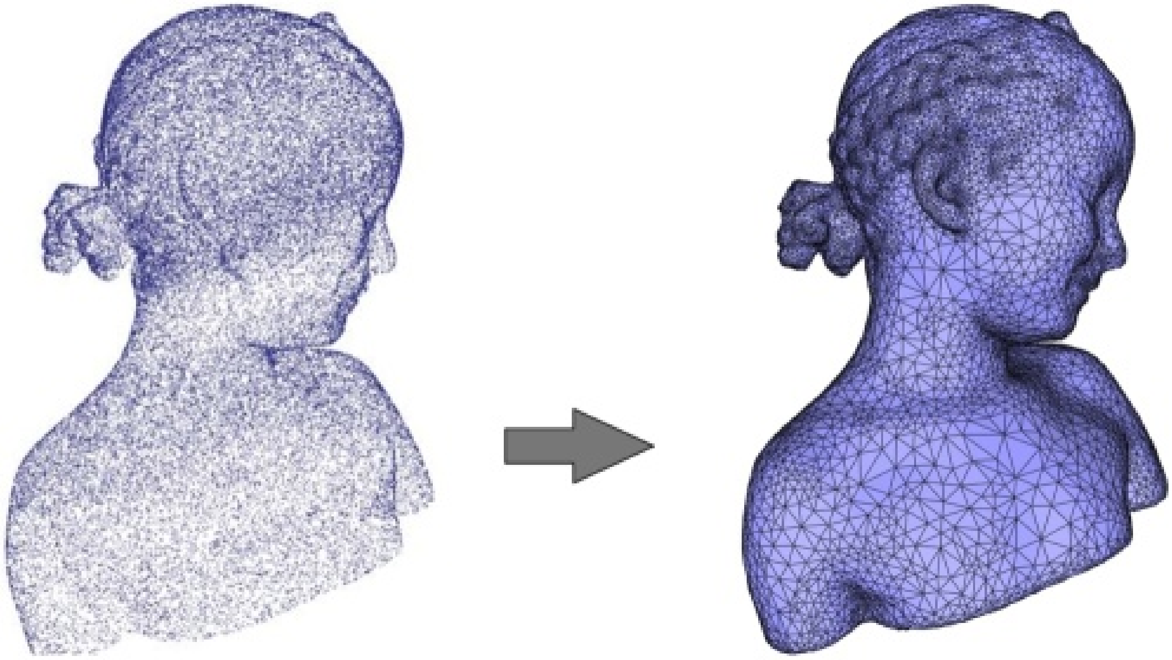


Figure 49.3: Poisson reconstruction. Left: 120K points sampled on a statue (Minolta laser scanner). Right: reconstructed surface mesh.

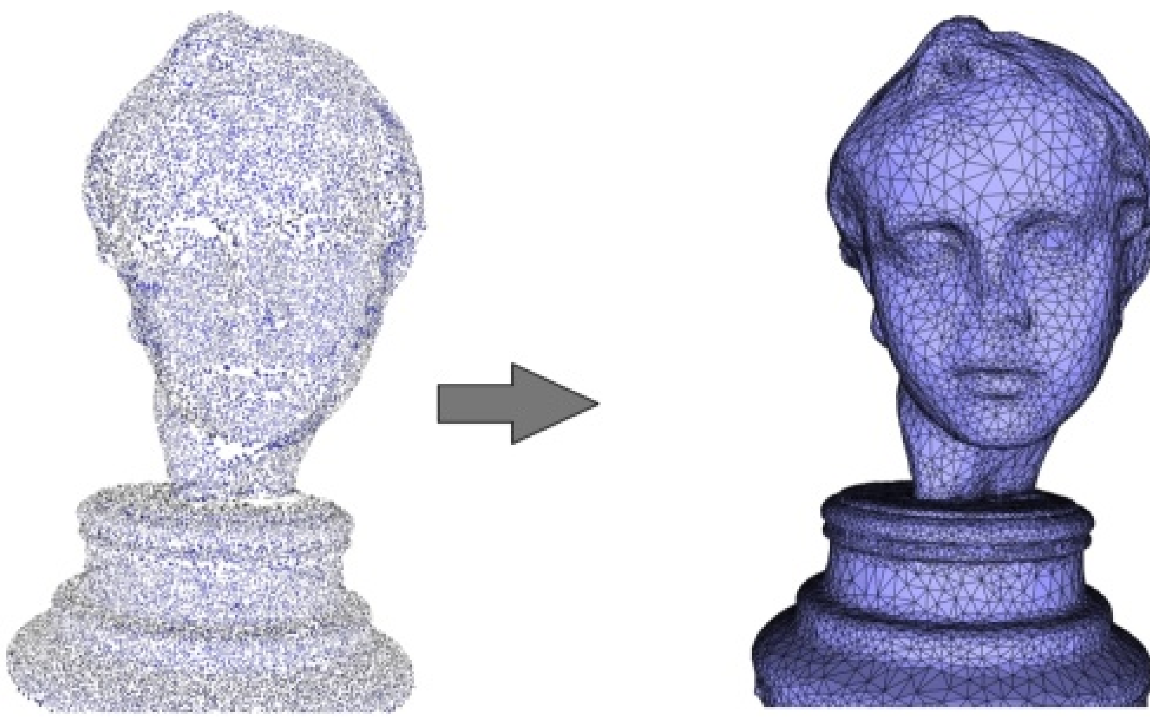


Figure 49.4: Left: 120K points sampled on a statue (Minolta laser scanner). Right: reconstructed surface mesh.

The algorithm is fairly robust to anisotropic sampling and to noise. It is also robust to missing data through filling the corresponding holes as the algorithm is designed to reconstruct the indicator function of an inferred solid (see Figure 49.5).

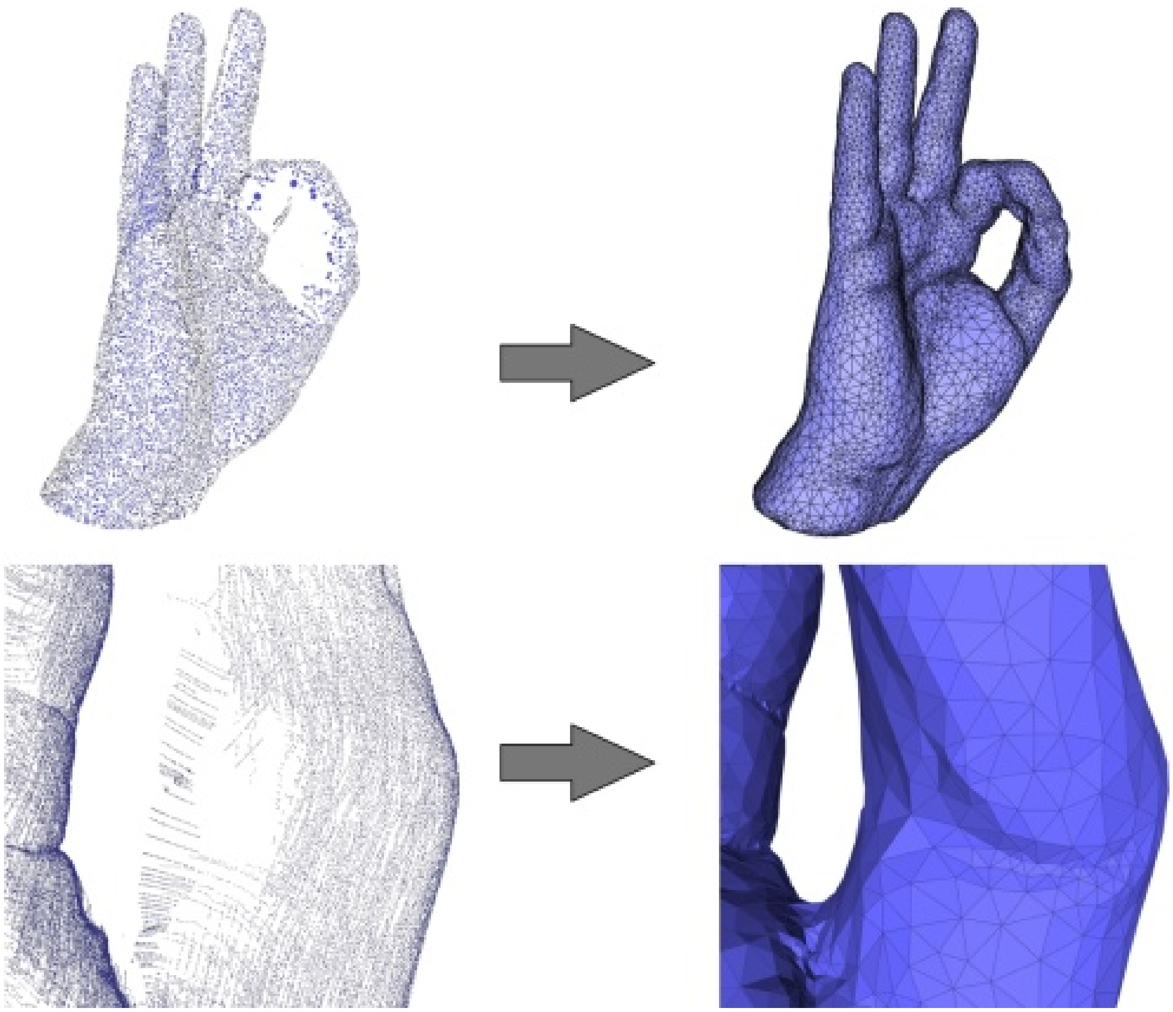


Figure 49.5: Top left: 65K points sampled on a hand (Kreon laser scanner). Bottom left: the point set is highly anisotropic due to the scanning technology. Right: reconstructed surface mesh and closeup. The holes are properly closed.

The algorithm is in general not robust to outliers, although a few outliers do not always create a failure, see Figure 49.6.

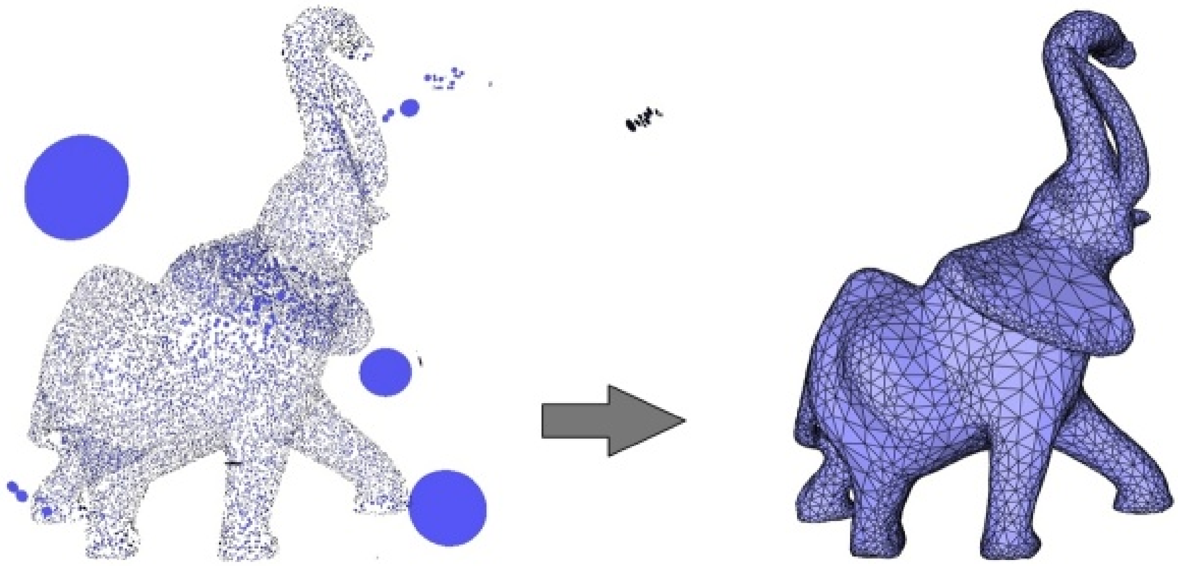


Figure 49.6: Left: 70K points sampled on an elephant with few outliers emphasized with disks. Right: reconstructed surface mesh.

The algorithm works well even when the inferred surface is composed of several connected components, provided that both all normals are properly estimated and oriented (the current CGAL normal orienter algorithm may fail in some cases, see *CGAL::mst_orient_normals()*), and that the final contouring algorithm is properly seeded for each component. When the inferred surface is composed of several nested connected components care should be taken to orient the normals of each component in alternation (inward/outward) so that the final contouring stage picks a proper contouring value.

Contouring Parameters

Our implementation of the Poisson surface reconstruction algorithm computes an implicit function represented as a piecewise linear function over the tetrahedra of a 3D Delaunay triangulation constructed from the input points then refined through Delaunay refinement. For this reason, any iso-surface is also piecewise linear and hence may contain sharp creases. As the contouring algorithm *CGAL::make_surface_mesh()* expects a smooth implicit function these sharp creases may create spurious clusters of vertices in the final reconstructed surface mesh when setting a small mesh sizing or surface approximation error parameter (see Figure 49.7).
One way to avoid these spurious clusters consists of adjusting the mesh sizing and surface approximation parameters large enough compared to the average sampling density (obtained through *CGAL::compute_average_spacing()*) so that the contouring algorithm "perceives" a smooth iso-surface. We recommend to use the following contouring parameters:

- Max triangle radius: at least 100 times the average spacing.
- Approximation distance: at least 0.25 times the average spacing.

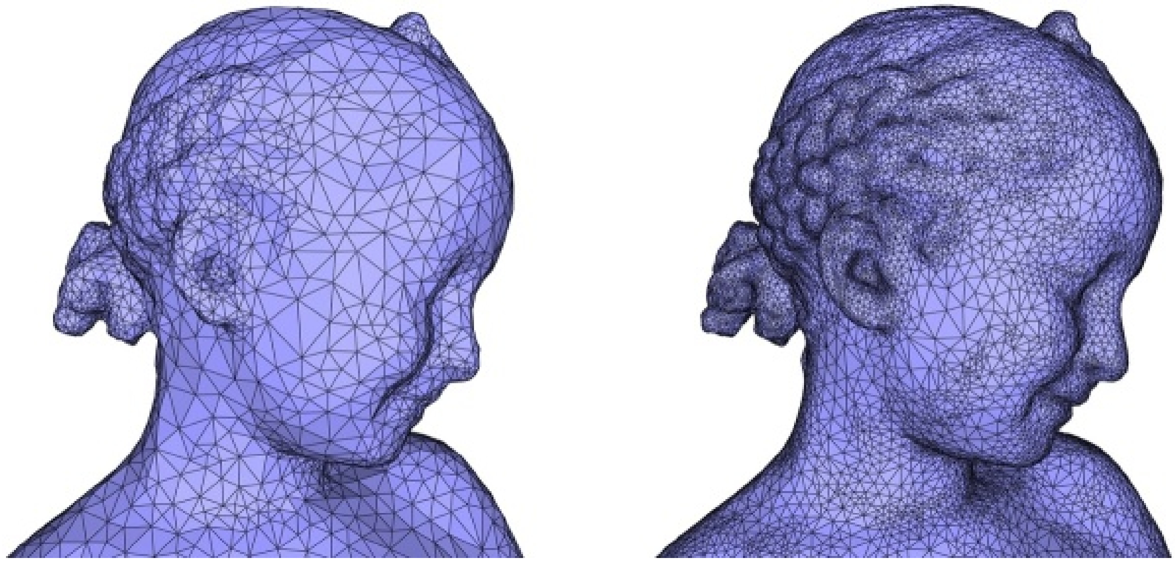


Figure 49.7: Left: surface reconstructed with approximation distance = 0.25 * average spacing. Right: surface reconstructed with approximation distance = 0.15 * average spacing. Notice the spurious cluster on the chick.

49.6.2 Degraded Conditions

The conditions listed above are rather restrictive and in practice not all of them are met in the applications. We now illustrates the behavior of the algorithm when the conditions are not met in terms of sampling, wrongly oriented normals, noise and sharp creases.

Sparse Sampling

The reconstruction algorithm expects a sufficiently dense point set. Although there is no formal proof of correctness of the algorithm under certain density conditions due to its variational nature, our experiments show that the algorithm reconstructs well all thin features when the local spacing is at most one tenth of the local feature size (the distance to the medial axis, which captures altogether curvature, thickness and separation). When this condition is not met the reconstruction does not reconstruct the thin undersampled features (see Figure 49.8).

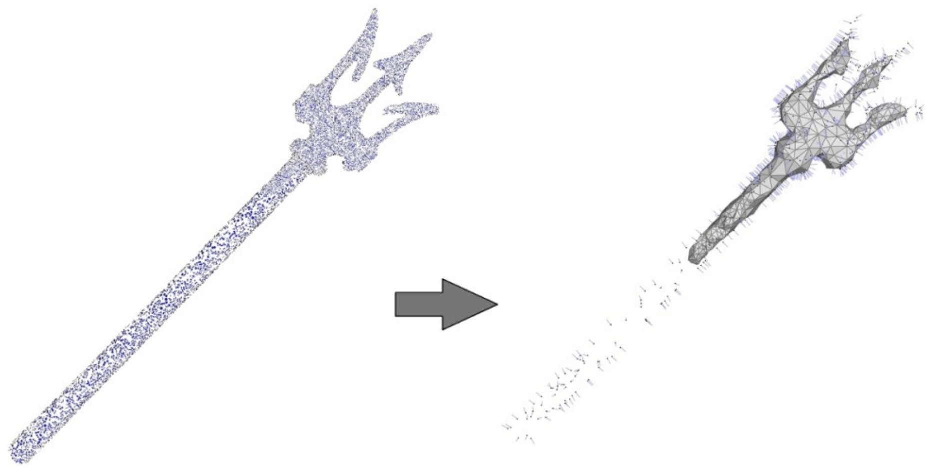


Figure 49.8: Left: 50K points sampled on the Neptune trident. The reconstruction (not shown) is successful in this case. Right: point set simplified to 1K points then reconstructed (all input points are depicted with normals). The thin feature is not reconstructed.

Large Holes

The reconstruction is devised to solve for an implicit function which is an approximate indicator function of an inferred solid. For this reason the contouring algorithm always extracts a closed surface mesh and hence is able to fill the small holes where data are missing due, e.g., to occlusions during acquisition. In case of large holes the algorithm still closes them all but sometimes in an unexpected manner. In addition the resulting piecewise linear implicit function may exhibit large triangle patches and sharp creases as the 3D Delaunay triangulation used for solving is very coarse where the holes are filled (see Figure 49.9).

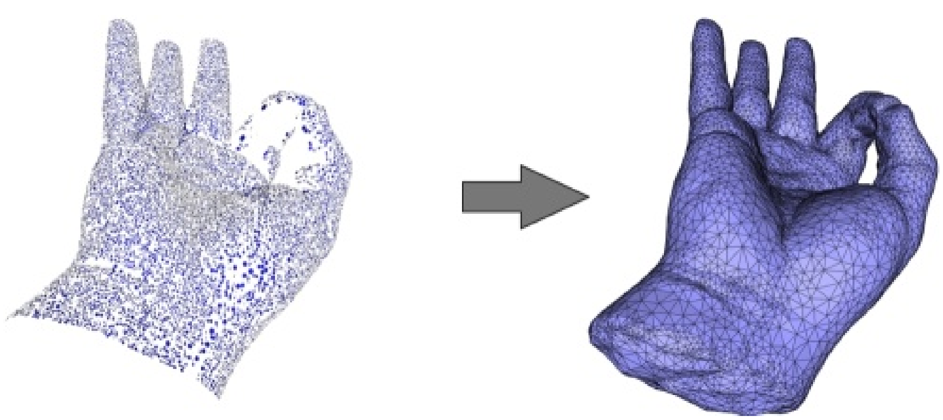


Figure 49.9: Left: 65K points sampled on a hand with no data captured at the wrist base. Right: reconstructed surface mesh. The surface is properly closed on the fingers and also closed at the wrist but in a less plausible manner.

Wrongly Oriented Normals

The Poisson surface reconstruction approaches solves for an implicit function whose gradient best matches a set of input normals. Because it solves this problem in the least squares sense, it is robust to few isolated wrongly oriented (flipped) normals. Nevertheless a cluster of wrongly oriented normals leads to an incorrect implicit function and hence to spurious geometric or even topological distortion (see Figure 49.10).

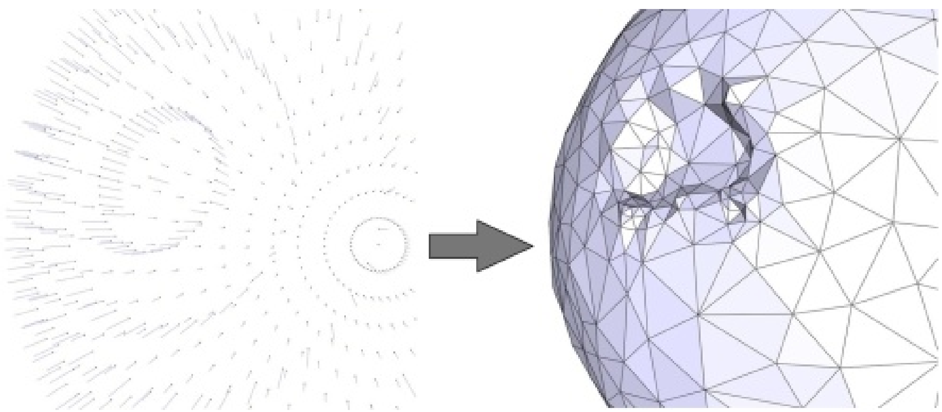


Figure 49.10: Left: points sampled on a sphere with a cluster of wrongly oriented normals. Right: reconstructed surface mesh with a spurious bump.

Noise and Outliers

A large amount of noise inevitably impacts on the reconstruction (see Figure 49.11, top) and the current implementation does not provide any mean to trade data fitting for smoothness. Nevertheless if the signal-to-noise ratio is sufficiently high and/or the surface approximation and sizing parameters set for contouring the iso-surface is large with respect to the noise level the output surface mesh will appear smooth (not shown). If the user wants to produce a smooth and detailed output surface mesh, we recommend to apply smoothing through *CGAL::jet_smooth_point_set()* ((see Figure 49.11, bottom).

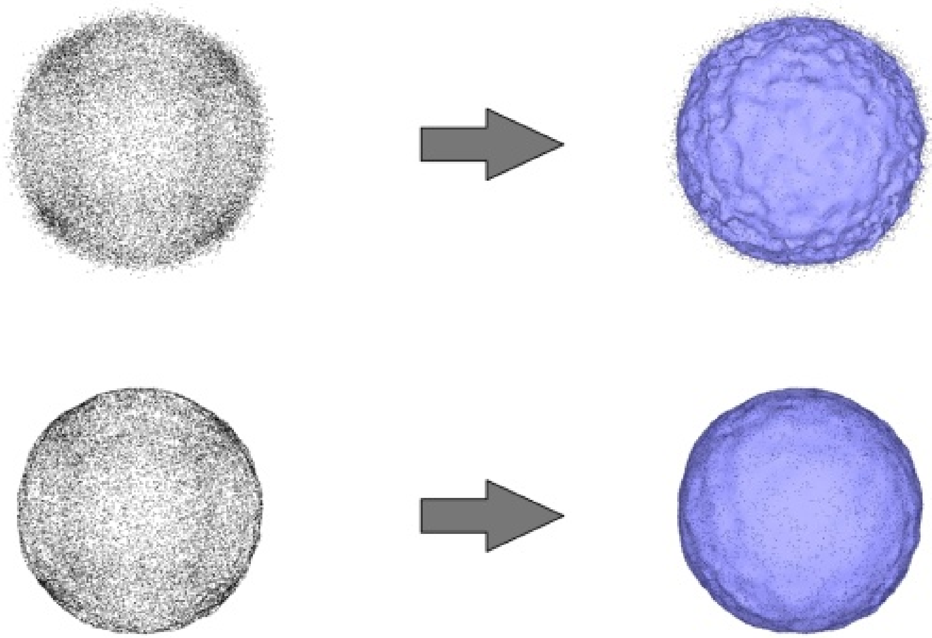


Figure 49.11: Top-left: points sampled on a sphere and corrupted with a lot of noise. Top-right: reconstructed surface mesh. Bottom-left: smoothed point set. Bottom-right: reconstructed surface mesh.

For a large number of outliers the failure cases (not shown) translate into spurious small connected components and massive distortion near the inferred surface. In this case the outliers must be removed through *CGAL::remove_outliers()*.

Sharp Creases

The current reconstruction algorithm is not able to recover the sharp creases and corners present in the inferred surface. This translates into smoothed sharp creases.

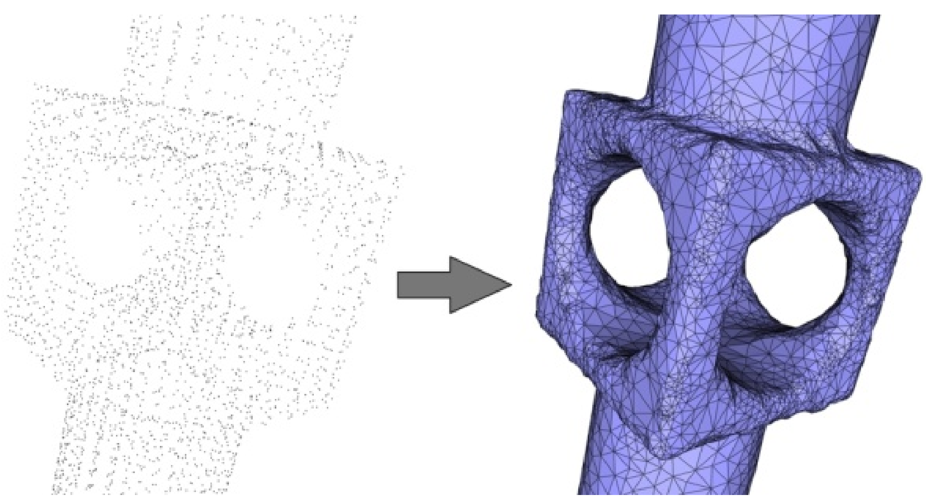


Figure 49.12: Left: 5K points sampled on a mechanical piece with sharp features (creases, darts and corners). Right: reconstructed surface mesh with smoothed creases.

Surface Reconstruction from Point Sets Reference Manual

Pierre Alliez, Laurent Saboret, Gael Guennebaud

This CGAL component implements a state-of-the-art surface reconstruction method: Poisson Surface Reconstruction. The input is an unorganized point set, possibly with attributes such as unoriented or oriented normals. The output is either an implicit function (ready for evaluation by any contouring algorithm), or a surface mesh generated by the CGAL surface mesh generator.

Note that since reconstruction methods often require pre-processing a point set (through reduction, smoothing, outlier removal, normal estimation, normal orientation), we provide components devoted to these tasks in the Point Set Processing component.

49.7 Classified Reference Pages

49.7.1 Classes

49.8 Alphabetical List of Reference Pages

Class

CGAL::Poisson_reconstruction_function<GeomTraits>

Definition

Given a set of 3D points with oriented normals sampled on the boundary of a 3D solid, the Poisson Surface Reconstruction method [KBH06] solves for an approximate indicator function of the inferred solid, whose gradient best matches the input normals. The output scalar function, represented in an adaptive octree, is then iso-contoured using an adaptive marching cubes.

Poisson_reconstruction_function implements a variant of this algorithm which solves for a piecewise linear function on a 3D Delaunay triangulation instead of an adaptive octree.

#include <CGAL/Poisson_reconstruction_function.h>

Parameters

template<class Gt>
class *Poisson_reconstruction_function*;

Parameters:
Gt: Geometric traits class.

Is Model for the Concepts

Model of the *ImplicitFunction* concept.

Types

Poisson_reconstruction_function<GeomTraits>:: Geom_traits

Geometric traits class.

Poisson_reconstruction_function<GeomTraits>:: FT	typedef to *Geom_traits::FT*
Poisson_reconstruction_function<GeomTraits>:: Point	typedef to *Geom_traits::Point_3*
Poisson_reconstruction_function<GeomTraits>:: Vector	typedef to *Geom_traits::Vector_3*
Poisson_reconstruction_function<GeomTraits>:: Sphere	typedef to *Geom_traits::Sphere_3*

Creation

template<typename InputIterator, typename PointPMap, typename NormalPMap>
Poisson_reconstruction_function<GeomTraits> fct(InputIterator first,
InputIterator beyond,
PointPMap point_pmap,

NormalPMap normal_pmap)

Creates a Poisson implicit function from the [first, beyond) range of points.
Template Parameters:
InputIterator: iterator over input points. *PointPMap*: is a model of *boost::ReadablePropertyMap* with a *value_type* = *Point_3*. It can be omitted if *InputIterator value_type* is convertible to *Point_3*. *NormalPMap*: is a model of *boost::ReadablePropertyMap* with a *value_type* = *Vector_3*.
Parameters:
first: iterator over the first input point. *beyond*: past-the-end iterator over the input points. *point_pmap*: property map to access the position of an input point. *normal_pmap*: property map to access the oriented normal of an input point.

Operations

Sphere *fct.bounding_sphere() const* Returns a sphere bounding the inferred surface.

template<class SparseLinearAlgebraTraits_d>
bool *fct.compute_implicit_function(SparseLinearAlgebraTraits_d solver = SparseLinearAlgebraTraits_d())*

The function *compute_implicit_function*() must be called after the insertion of oriented points. It computes the piecewise linear scalar function operator() by: applying Delaunay refinement, solving for operator() at each vertex of the triangulation with a sparse linear solver, and shifting and orienting operator() such that it is 0 at all input points and negative inside the inferred surface.
Template parameters:
SparseLinearAlgebraTraits_d: Symmetric definite positive sparse linear solver. If EIGEN 3.1 (or greater) is available and *CGAL_EIGEN3_ENABLED* is defined, the default solver is *Eigen::ConjugateGradient*.
Returns: false if the linear solver fails.
Parameters:
solver: sparse linear solver.

FT *fct.operator()(const Point& p) const*

ImplicitFunction interface: evaluates the implicit function at a given 3D query point. The function *compute_implicit_function* must be called before the first call to *operator()*.

Point *fct.get_inner_point() const* Returns an arbitray point located inside the inferred surface.

Example

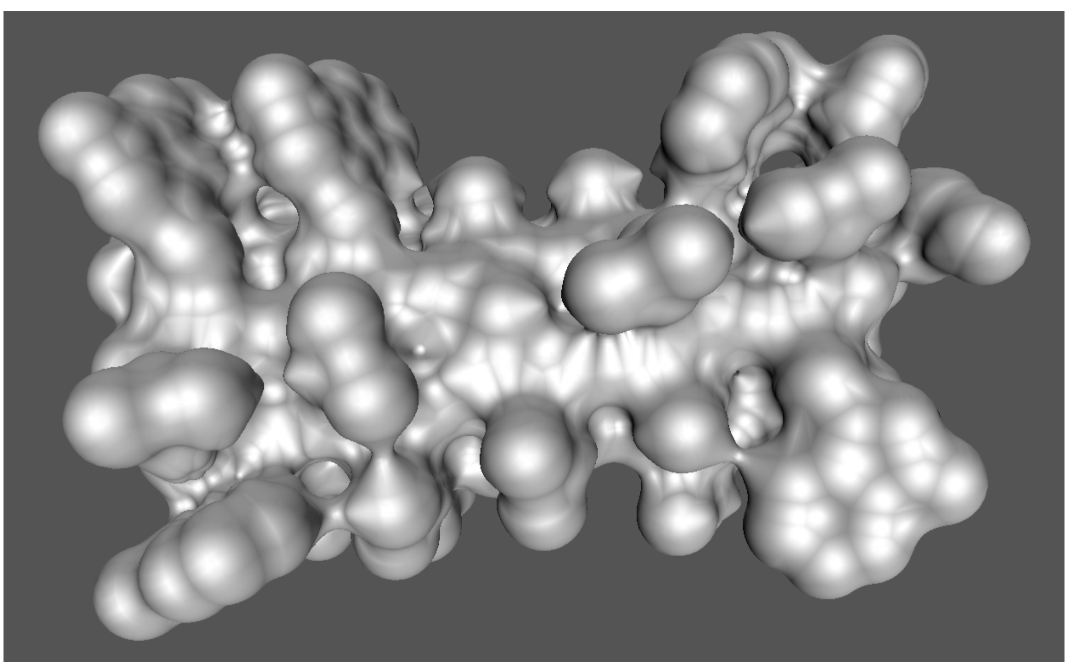
See *Surface_reconstruction_points_3/poisson_reconstruction_example.cpp*.

Chapter 50

3D Skin Surface Meshing

Nico Kruithof

Contents



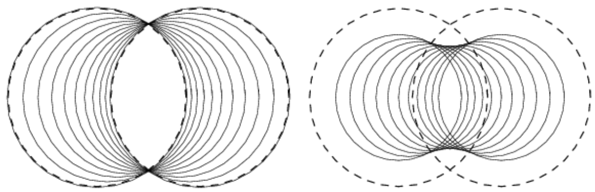


Figure 50.1: Left: Convex combinations of two weighted points (the two dashed circles). Right: The skin curve of the weighted points. The smaller circles form a subset of the weighted points whose boundary is the skin curve.

50.1 Introduction

Skin surfaces, introduced by Edelsbrunner in [Ede99], have a rich and simple combinatorial and geometric structure that makes them suitable for modeling large molecules in biological computing. Meshing such surfaces is often required for further processing of their geometry, like in numerical simulation and visualization.

A skin surface is defined by a set of weighted points (input balls) and a scalar called the shrink factor. If the shrink factor is equal to one, the surface is just the boundary of the union of the input balls. For a shrink factor smaller than one, the skin surface becomes tangent continuous, due to the appearance of patches of spheres and hyperboloids connecting the balls.

This package constructs a mesh isotopic to the skin surface defined by a set of balls and a shrink factor using the algorithm described in [KV05].

An optimized algorithm is implemented for meshing the union of a set of balls.

50.2 Definition of a Skin Surface

This section first briefly reviews skin surfaces. For a more thorough introduction to skin surfaces, we refer to [Ede99] where they were originally introduced.

A skin surface is defined in terms of a finite set of weighted points $\mathbf{P}$ and a shrink factor s, with $0 \leq s \leq 1$. A weighted point $\mathbf{p} = (p, w_p) \in \mathbb{R}^3 \times \mathbb{R}$ corresponds to a ball with center p and radius $\sqrt{w_p}$. A weighted point with zero weight is called an unweighted point.

A pseudo distance between a weighted point $\mathbf{p} = (p, w_P)$ and an unweighted point x is defined as

$$\pi(\mathbf{p}, x) = \| p - x \|^2 - w_p,$$

where $\| p - x \|$ is the Euclidean distance between p and x. The ball corresponding to a weighted point $\mathbf{p}$ is the zero set of $\pi(\mathbf{p}, \cdot)$. Note that if $w_p < 0$ the radius of the ball is imaginary and the zero-set is empty.

We can take convex combinations of weighted points by taking convex combinations of their distance functions. Figure 50.1 (left) shows weighted points that are obtained as convex combinations of the dashed circles. For further reading on the space of circles and spheres we refer to [Ped70].

Starting from a weighted point $\mathbf{p} = (p, w_P)$, the shrunk weighted point $\mathbf{p}^s$ is obtained by taking a convex combination with the unweighted point centered at p, formally $\mathbf{p}^s = s\mathbf{p} + (1-s)\mathbf{p}'$, with $\mathbf{p}' = (p, 0)$. A simple calculation shows that, $\mathbf{p}^s = (p, s \cdot w_p)$. The set $\mathbf{P}^s$ is the set obtained by shrinking every weighted point of $\mathbf{P}$ by a factor s: $\mathbf{P}^s = \{\mathbf{p}^s \mid \mathbf{p} \in \mathbf{P}\}$. The shrunk weighted points of Figure 50.1 (left) are shown in Figure 50.1 (right).

We now define the skin surface $\mathrm{skn}^s(\mathbf{P})$ associated with a set of weighted points $\mathbf{P}$. Consider the set of weighted points obtained by taking the convex hull of the input weighted points. A calculation shows that every weighted point lies within the union of the input balls. Next, we shrink each weighted point in the convex hull with the shrink factor s. Hence, we multiply the radius of the corresponding (real) input circles with a factor $\sqrt{s}$. The skin surface is the boundary of the union of this set of weighted points:

$$\mathrm{skn}^s(\mathbf{P}) \quad = \quad \partial \bigcup \{\mathbf{p}^s \mid \mathbf{p} \in \mathrm{conv}(\mathbf{P})\}. \tag{50.1}$$

Here $\mathrm{conv}(\mathbf{P}) \subset \mathbb{R}^3 \times \mathbb{R}$ is the convex hull of a set of weighted points $\mathbf{P}$, whereas ∂ denotes the boundary – in $\mathbb{R}^3$ – of the union of the corresponding set of balls.

Recall that each weighted point in the convex hull of the input weighted points is contained in the union of the input weighted points. Hence, for a shrink factor equal to one, the skin surface is the boundary of the union of the input weighted points.

By definition of a skin surface, the weights of the input balls (their radius-squared) are shrunk with a factor of s and the skin surface wraps around the shrunk input balls. In order to make the skin surface wrap around the (unshrunk) input balls, we can first increase the weights of the input balls by multiplying them with a factor $1/s$ and then compute the skin surface.

50.3 The Interface

The interface to the skin surface package consists of one main function, taking a set of weighted points and a shrink factor and outputting the meshed surface. Further, it defines classes and functions and classed used to perform the main steps of the algorithm. There are two global classes *Skin_surface_3* and *Union_of_balls_3* both of which are models of the concept *SkinSurface_3* and there are two functions to extract a mesh of the skin surface (union of balls) from the objects of the aforementioned classes. A final function takes a mesh and the *Skin_surface_3* (*Union_of_balls_3*) object it is constructed from and refines the mesh. This section describes these classes and functions in more detail.

The main function of the skin surface package takes an iterator range of weighted points, a shrink factor and the number of subdivision steps and outputs a mesh in a *CGAL::Polyhedron_3*:

```
template <class WP_iterator, class Polyhedron_3>
void    make_skin_surface_mesh_3( Polyhedron_3 &p,
                                  WP_iterator begin,
                                  WP_iterator end,
                                  FT shrink_factor=.5,
                                  int nSubdiv=0,
                                  bool grow_balls = true)
```

Where, *FT* is the number type used by the *Weighted_points*.

To obtain more control over the algorithm, the different steps can also be performed separately. First, a *Skin_surface_3* object is created from an iterator range of weighted points and a shrink factor. Optional arguments are a boolean telling whether the input weighted points should be grown in such a way that the skin surface wraps around the input balls instead of the shrunk input balls.

```
template <class SkinSurfaceTraits_3>
        Skin_surface_3( WP_iterator begin, WP_iterator end, FT shrink_factor, bool grow_balls = true)
```

The template parameter should implement the *SkinSurfaceTraits_3* concept. The type *WP_iterator*, is an iterator over weighted points as defined by *SkinSurfaceTraits_3* and *FT* is the number type used by the weighted points.

For a shrink factor equal to one the skin surface is the boundary of the union of the input balls. In this case the algorithm used for meshing the skin surface greatly simplifies. These optimizations are implemented in the class *Union_of_balls_3*. The constructor for the union of balls class is similar, except for the missing shrink factor:

template <class SkinSurfaceTraits_3>
Union_of_balls_3(WP_iterator begin, WP_iterator end, bool grow_balls = true)

With a model of the concept *SkinSurface_3* it is possible to generate a coarse mesh isotopic to the skin surface. Using the function *mesh_skin_surface_3* with signature:

template <class SkinSurface_3, class Polyhedron>
void mesh_skin_surface_3(SkinSurface_3 skin_surface, Polyhedron &p)

The last function takes the (coarse) mesh and subdivides it in-situ by applying a given number of 1-4 split operations (each triangle is split into four sub-triangles) and moving the new vertices towards the skin surface. If the number of iterations is not specified, one subdivision step is done. The object of the *SkinSurface_3* concept used to construct the coarse mesh is needed to move new points on the skin surface.

template <class SkinSurface_3, class Polyhedron >
void subdivide_skin_surface_mesh_3(SkinSurface_3 skinsurface, Polyhedron &p, int iterations=1)

50.4 Timings

The timings of the construction of the coarse mesh and the first subdivision are given in seconds and were done on a Pentium 4, 3.5 GHz, with 1 Gb of memory.

Data set	Number of weighted points	Coarse mesh	first subdivision step
Caffeine	23	0.2 sec.	0.05 sec.
Gramicidin A	318	5 sec.	2 sec.

50.5 Example Programs

50.5.1 Meshing a Skin Surface

The following example shows the construction of a coarse mesh of the skin surface using the function *make_skin_surface_mesh_3*. The output is a *CGAL::Polyhedron*.

```
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/make_skin_surface_mesh_3.h>
#include <list>

typedef CGAL::Exact_predicates_inexact_constructions_kernel K;
typedef K::Point_3                                          Bare_point;
```

```
typedef CGAL::Weighted_point<Bare_point,K::RT>
    Weighted_point;
typedef CGAL::Polyhedron_3<K>                               Polyhedron;

int main() {
  std::list<Weighted_point> l;
  double                    shrinkfactor = 0.5;

  l.push_front(Weighted_point(Bare_point( 1,-1,-1), 1.25));
  l.push_front(Weighted_point(Bare_point( 1, 1, 1), 1.25));
  l.push_front(Weighted_point(Bare_point(-1, 1,-1), 1.25));
  l.push_front(Weighted_point(Bare_point(-1,-1, 1), 1.25));

  Polyhedron p;

  CGAL::make_skin_surface_mesh_3(p, l.begin(), l.end(), shrinkfactor);

  return 0;
}
```

File: examples/Skin_surface_3/skin_surface_simple.cpp

50.5.2 Meshing and Subdividing a Skin Surface

The following example shows the construction of mesh of the skin surface by explicitly performing different steps of the algorithm. It first constructs a *Skin_surface_3* object from an iterator range of weighted points and a shrink factor. From this object, the coarse mesh isotopic to the skin surface is extracted using the function *CGAL::mesh_skin_surface_3*

Next, the coarse mesh is refined to obtain a better approximation. The use of *CGAL::Skin_surface_polyhedral_items_3<Skin_surface_3>* in the *CGAL::Polyhedron* is not necessary, but gives the subdivision a significant speedup.

```
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/Skin_surface_3.h>
#include <CGAL/Polyhedron_3.h>
#include <CGAL/mesh_skin_surface_3.h>
#include <CGAL/subdivide_skin_surface_mesh_3.h>
#include "skin_surface_writer.h"
#include <list>
#include <fstream>

typedef CGAL::Exact_predicates_inexact_constructions_kernel K;
typedef CGAL::Skin_surface_traits_3<K>                      Traits;
typedef CGAL::Skin_surface_3<Traits>
    Skin_surface_3;
typedef Skin_surface_3::FT                                  FT;
typedef Skin_surface_3::Weighted_point
    Weighted_point;
typedef Weighted_point::Point                               Bare_point;
typedef CGAL::Polyhedron_3<K,
  CGAL::Skin_surface_polyhedral_items_3<Skin_surface_3> >   Polyhedron;
```

```
int main() {
  std::list<Weighted_point> l;
  FT                        shrinkfactor = 0.5;

  l.push_front(Weighted_point(Bare_point( 1,-1,-1), 1.25));
  l.push_front(Weighted_point(Bare_point( 1, 1, 1), 1.25));
  l.push_front(Weighted_point(Bare_point(-1, 1,-1), 1.25));
  l.push_front(Weighted_point(Bare_point(-1,-1, 1), 1.25));

  Polyhedron p;

  Skin_surface_3 skin_surface(l.begin(), l.end(), shrinkfactor);
  CGAL::mesh_skin_surface_3(skin_surface, p);

  CGAL::subdivide_skin_surface_mesh_3(skin_surface, p);

  std::ofstream out("mesh.off");
  out << p;

  return 0;
}
```

File: examples/Skin_surface_3/skin_surface_subdiv.cpp

3D Skin Surface Meshing Reference Manual

Nico Kruithof

Skin surfaces form a class of tangent continuous surfaces defined by a set of balls and a scalar called the shrink factor. The skin surface package constructs an triangular mesh isotopic to the skin surface, using the algorithm described in [KV05]. The mesh is stored in a *CGAL::Polyhedron_3*.

Concepts

Classes

Functions

50.6 Alphabetical List of Reference Pages

SkinSurface_3

Definition

The concept SkinSurface_3 defines a skin surface and provides an interface for the dedicated skin surface mesher. The concept requires a constructor from an iterator range of weighted points and a shrink factor. By default the input balls are grown in such that the skin surface wraps around the input balls.

Has Models

CGAL::Skin_surface_3<SkinSurfaceTraits_3>
CGAL::Union_of_balls_3<SkinSurfaceTraits_3>

Types

SkinSurface_3:: Geometric_traits	The geometric traits used for the construction of the regular triangulation.
SkinSurface_3:: Weighted_point	The *Weighted_point* type defined by the *Geometric_traits*.
SkinSurface_3:: Bare_point	The *Bare_point* type defined by the *Geometric_traits*.
SkinSurface_3:: FT	The *FT* type defined by the *Geometric_traits*. This is the number type used by the weighted points.

Creation

SkinSurface_3 skin(WP_iterator begin, WP_iterator end, RT shrink_factor);

The mandatory arguments to the constructor are an iterator range *[begin,end)* of weighted points and a shrink factor between 0 and 1.

Operations

template <class Polyhedron_3>
void skin.mesh_skin_surface_3(Polyhedron_3 &p)

Constructs a coarse mesh in *p*.
Precondition: *Polyhedron_3::HDS* can be used as the template argument of the *Polyhedron_incremental_builder_3< HDS>*.

template <class Polyhedron_3>
void skin.subdivide_skin_surface_mesh_3(Polyhedron_3 &p, int nSubdiv = 1)

Subdivides the skin surface using *nSubdiv* 1-4 split operations (each triangle is split into four sub-triangles) and the new vertices are moved towards the skin surface.

Concept

SkinSurfaceTraits_3

Required types and member functions for the SkinSurfaceTraits_3 concept. This geometric traits concept is used for the construction of a polyhedral mesh approximating a skin surface *CGAL::skin_surface_3<Traits>*.

Refines

RegularTriangulationTraits_3

Types

SkinSurfaceTraits_3:: RT The number type.

SkinSurfaceTraits_3:: Construct_anchor_point_3

A constructor object which provides the following function operator:
Point operator()(const Point &center_del, const Point &center_vor);
Constructs the anchor point in between the Delaunay and Voronoi centers, i.e., the point $(1-s)\cdot center\_del + s\cdot center\_vor$, where s is the shrink factor.

Creation

SkinSurfaceTraits_3 traits(RT s=.5);

constructor that takes the shrink factor as argument. For meshing the boundary of the union of a set of balls, the shrink factor is discarded.

Has Models

CGAL::Skin_surface_traits_3<K> . . . . page 3123

Operations

The following functions give access to the constructor objects:

Regular_RT *traits.shrink_factor() const*

Returns the shrink factor.

Construct_anchor_point_3 *traits.construct_anchor_point_3_object() const*

returns a *Construct_anchor_point_3* object.

See Also

CGAL::Skin_surface_3<SkinSurfaceTraits_3> . . . . page 3121
CGAL::Skin_surface_traits_3<K> . . . . page 3123

CGAL::Skin_surface_3<SkinSurfaceTraits_3>

Class

Definition

The *Skin_surface_3<SkinSurfaceTraits_3>* is the main class in the *Skin_surface_3* package. It is a model of the concept *SkinSurface_3*.

The template argument is required to be a model of the concept *SkinSurfaceTraits_3*, which means that it provides predicates to construct a regular triangulation of the weighted points and for point location in the mixed complex.

#include <CGAL/Skin_surface_3.h>

Is Model for the Concepts

SkinSurface_3

Class

CGAL::Union_of_balls_3<SkinSurfaceTraits_3>

Definition

The *Union_of_balls_3<SkinSurfaceTraits_3>* is used to represent a skin surface with shrink factor equal to one, which is the boundary of the union of the input balls. This case is handled separately since the mixed complex is equal to the power diagram, which has a much simpler structure.

The template argument must be a model of the concept *SkinSurfaceTraits_3*, which means that it provides the predicates to construct a regular triangulation of the weighted points.

#include <CGAL/Union_of_balls_3.h>

Is Model for the Concepts

SkinSurface_3

CGAL::Skin_surface_traits_3<K>

Class

Definition

The class *Skin_surface_traits_3<K>* is designed as a default traits class for the class *Skin_surface_3< SkinSurfaceTraits_3>*. It is a model of the *SkinSurfaceTraits_3*.

The template argument *K* must be a model of the *Kernel* concept.

#include <CGAL/Skin_surface_traits_3.h>

Is Model for the Concepts

SkinSurfaceTraits_3

Function

CGAL::make_skin_surface_mesh_3<Polyhedron_3>

Definition

The function *make_skin_surface_mesh_3<Polyhedron_3>* constructs a mesh isotopic to the skin surface based on the algorithm in [KV05]. It takes as input a range of weighted points and a shrink factor and outputs the mesh in a *Polyhedron_3* object. A number of subdivision steps might be applied to refine the mesh.

#include <CGAL/make_skin_surface_mesh_3.h>

void *mesh_skin_surface_3<Polyhedron_3>(Polyhedron_3 &p,*
WP_iterator begin,
WP_iterator end,
double shrink_factor=.5,
int nSubdivisions=0,
bool grow_balls=true)

Constructs a mesh of the skin surface defined by the weighted points and the shrink factor.
Precondition: *Polyhedron_3::HDS* can be used as the template argument of the *Polyhedron_incremental_builder_3<HDS>*.

CGAL::mesh_skin_surface_3<SkinSurface_3, Polyhedron_3>

Function

Definition

The function *mesh_skin_surface_3<SkinSurface_3, Polyhedron_3>* constructs a mesh isotopic to the skin surface based on the algorithm in [KV05]. It takes as input a *SkinSurface_3* object, which is a model of the *SkinSurface_3* concept and outputs the mesh in a *Polyhedron_3* object.

#include <CGAL/mesh_skin_surface_3.h>

void *make_skin_surface_mesh_3<SkinSurface_3,Polyhedron_3>(SkinSurface_3 skin_surface, Polyhedron_3 &p)*

Constructs a mesh of the *skin_surface* in *p*.
Precondition: *SkinSurface_3* is a model of the concept *SkinSurface_3* and *Polyhedron_3::HDS* can be used as the template argument of the *Polyhedron_incremental_builder_3<HDS>*.

Function

CGAL::subdivide_skin_surface_mesh_3<SkinSurface_3, Polyhedron_3>

Definition

The function *subdivide_skin_surface_mesh_3<SkinSurface_3, Polyhedron_3>* subdivides a skin surface mesh constructed by the function *mesh_skin_surface_3<SkinSurface_3, Polyhedron_3>*.

#include <CGAL/subdivide_skin_surface_mesh_3.h>

void subdivide_skin_surface_mesh_3<SkinSurface_3,Polyhedron_3>(SkinSurface_3 skin_surface,
Polyhedron_3 &p,
int nSubdiv = 1)

Subdivides the skin surface using *nSubdiv* 1-4 split operations (each triangle is split into four sub-triangles) and the new vertices are moved towards the skin surface.

Chapter 51

3D Mesh Generation

Pierre Alliez, Laurent Rineau, Stéphane Tayeb, Jane Tournois, Mariette Yvinec

Contents

51.1 Introduction

This package is devoted to the generation of isotropic simplicial meshes discretizing 3D domains. The domain to be meshed is a region of 3D space that has to be bounded. The region may be connected or composed of multiple components and/or subdivided in several subdomains.

Boundary and subdivision surfaces are either smooth or piecewise smooth surfaces, formed with planar or curved surface patches. Surfaces may exhibit 1-dimensional features (e.g. crease edges) and 0-dimensional features (e.g. singular points as corners tips, cusps or darts), that have to be fairly approximated in the mesh.

The output mesh is a 3-dimensional triangulation, including subcomplexes that approximate each input domain feature: subdomain, boundary surface patch or input domain feature with dimension 0 or 1. Thus, the output mesh includes a 3D submesh covering each subdomain, a surface mesh approximating each boundary or subdividing surface patch, a polyline approximation for each 1-dimensional feature and of course a vertex on each corner.

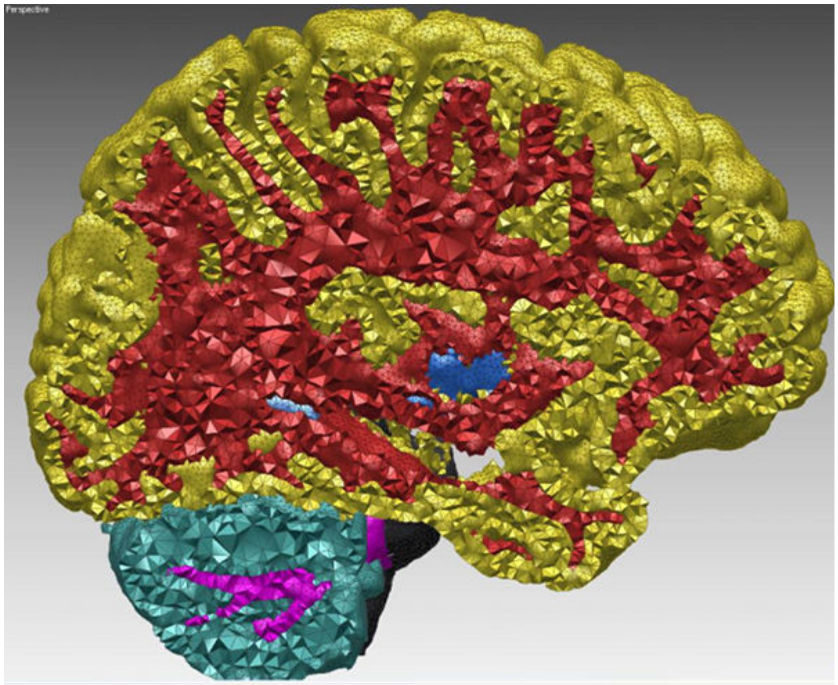


Figure 51.1: Cut-view of a multi-domain 3D mesh generated from a segmented image.

The main entry points of the package are two global functions that respectively generate and refine such meshes. The mesh generator is customized to output a mesh that fits as much as possible the user needs, for instance in terms of sizing field or with respect to some user customized quality criteria.

The meshing engine used in this mesh generator is based on Delaunay refinement [Che93, Rup95, She98b]. It uses the notion of restricted Delaunay triangulation to approximate 1-dimensional curve segments and surface patches [BO05]. Before the refinement, a mechanism of protecting balls is set up on 1-dimensional features, if any, to ensure a fair representation of those features in the mesh, and also to guarantee the termination of the refinement process, whatever may be the input geometry, in particular whatever small angles the boundary and subdivision surface patches may form [CDL07, CDR07]. The Delaunay refinement is followed by a mesh optimization phase to remove slivers and provide a good quality mesh.

Input domain

The domain to be meshed is assumed to be bounded and representable as a pure 3D complex. A 3D complex is a set of faces with dimension 0, 1, 2 and 3 such that all faces are pairwise interior disjoint, and the boundary of each face of the complex is the union of faces of the complex. The 3D complex is pure, meaning that each face is included in a face of dimension 3, so that the complex is entirely described by the set of its 3D faces and their subfaces. However the 3D complex needs not be connected. The set of faces with dimension lower or equal than 2 forms a 2D subcomplex which needs not be manifold, neither pure, nor connected: some 3D faces may have dangling 2D or 1D faces in their boundary faces.

In the rest of the documentation, we will refer to the input 3D complex as the input domain. The faces of the input domain with dimension 0, 1, 2 and 3 are called respectively *corners*, *curve segments*, *surface patches* and *subdomains* to clearly distinguish them from the faces of the mesh that are called vertices, edges, facets and cells.

Note that the input complex faces are not required to be linear nor smooth. Surface patches, for instance, may be smooth surface patches, or portions of surface meshes with boundaries. Curve segments may be for instance straight segments, curved segments or polylines. Each of those features will be accurately represented in the final mesh.

The 0 and 1-dimensional features of the input domain are usually singular points of the subdomain boundaries, however this is not required. Furthermore those features are not required to cover all the subdomains boundaries singularities but only those that need to be accurately represented in the final mesh. In the following, we say that a domain has *features* when it has 0 and 1-dimensional features that need to be accurately represented in the mesh, and we call those features *exposed features*. Therefore, a domain may be without features either because all boundary surface patches are smooth closed surfaces, or simply because the curves joining different surface patches and the singularities of those patches need not be accurately approximated in the final mesh.

Note also that input complex faces are not required to be connected. Faces of the input domain are identified by indexes. If a subdomain is not connected, its different components receive the same index. Likewise different surface patches, segment curves or corners may share the same index. Each connected component of a feature will be accurately represented in the final mesh. Note however that the occurrence of multiply connected faces in the input complex may affect the relevance of internal topological checks performed by the mesh generator. Also the mesh generator will not be able to apply different meshing criteria, e.g. different sizing field, for the different connected components of a single feature.

The domain is input to the mesh generation function, as a domain class, often called the oracle, that provides predicates and constructors related to the domain, the subdomains, the boundary surface patches and also the 0 and 1-dimensional exposed features, if any. Mainly, the oracle provides a predicate to test if a given query point belongs to the domain or not and to find in which subdomain it lies in the affirmative case. The domain class also provides predicates and constructors to test the intersection of a query line segment with the boundary surface patches and to build some intersection points if any. Lastly, if the input domain includes 1-dimensional exposed features, the domain class provides a way to construct sample points on these features.

The current implementation provides classes to represent domains bounded by isosurfaces of implicit functions, polyhedral domains and domains defined through 3D labeled images. Currently, 1-dimensional features may be defined as segments and polyline segments.

Output mesh

The resulting mesh is output as a subcomplex of a 3D Delaunay triangulation, in a class providing various iterators on mesh elements.

The 3D triangulation provides approximations of the subdomains, surface patches and curve segments and corners, according to the restricted Delaunay triangulation paradigm. This means that each subdomain is approximated by the union of the tetrahedral cells whose circumcenters are located inside the domain (or subdomain). Each surface patch is approximated by the union of the Delaunay mesh facets whose dual Voronoi edges intersect the surface patch. Such mesh facets are called *surface facets* in the following. The 1-dimensional exposed features are approximated by sequences of mesh edges and the 0-dimensional exposed features are represented by mesh vertices.

Delaunay Refinement

The mesh generation algorithm is mainly a Delaunay refinement process. The Delaunay refinement is preceded by a protecting phase to ensure an accurate representation of 1-dimensional features if any, and followed by an optimization phase to achieve a good quality mesh.

The Delaunay refinement process is driven by criteria concerning either the size and shape of mesh cells and surface facets. The refinement process terminates when there are no more mesh cells or surface facets violating the criteria.

The criteria are designed to achieve a nice spread of the mesh vertices while ensuring the termination of the refinement process. Those criteria may be somehow tuned to the user needs to achieve for instance the respect of a sizing field by mesh elements, some topological conditions on the representation of boundary surfaces in the mesh, and/or some error bound for the approximation of boundary surfaces. To some extend, the user may tune the Delaunay refinement to a prescribed trade-off between mesh quality and mesh density. The mesh density refers to the number of mesh vertices and cells, i.e. to the complexity of the mesh. The mesh quality referred to here is measured by the radius edge ratio of surface facets end mesh cells, where the radius edge ratio of a simplex (triangle or tetrahedron) is the the ratio between its circumradius and its shortest edge.

Protection of 0- and 1-dimensional exposed features

If the domain description includes 0 dimensional features, the corresponding points are inserted into the Delaunay triangulation from the start.

If the domain has 1-dimensional exposed features, the method of protecting balls [CDR07, CDL07] is used to achieve an accurate representation of those features in the mesh and to guarantee that the refinement process terminates whatever may be the dihedral angles formed by input surface patches incident to a given 1-feature or the angles formed by two 1-features incident to a 0-feature.

According to this method, the 1-dimensional features are sampled with points and covered by protecting balls centered on the sample points, in such a way that :
-no three balls intersect
-no pair of balls centered on different 1-features intersect.

The triangulation embedding the mesh is in fact a weighted Delaunay triangulation, and the triangulation is initialized by the insertion of all the protecting balls, regarded as weighted points. The Delaunay refinement process is then launched as before except that refinement points are no longer circumcenters but are weighted circumcenters. All Steiner vertices inserted by the refinement process are given a zero weight.

The method guarantees:
1) that each segment joining two successive centers on a 1-dimensional feature will stay in the triangulation, thus ensuring an accurate approximation of the 1-dimensional features.
2) that the refinement process will never try to insert a refinement point in the union of the protecting balls, which ensures the termination of the refinement process.

Optimization phase

Any tetrahedron that is quasi degenerate has a big radius edge ratio except those belonging to the family of slivers. A sliver is easily obtained as the convex hull of 4 points close to the equatorial circle of a 3D ball and roughly equally spread along this circle. The Delaunay refinement tracks tetrahedra with big radius edge ratio and therefore eliminates all kinds of badly shaped tetrahedra except slivers.

Therefore, at the end of the refinement process, some sliver shaped tetrahedra may occur in the mesh. The optimization phase aims at eliminating slivers.

The optimization phase is a succession of optimization processes, including possibly a Lloyd smoothing, an odt-smoothing, a perturber and an exuder.

The Lloyd and odt-smoother are global optimizers moving the mesh vertices to minimize a mesh energy. Those optimizers are described respectively in [DFG99, DW02] and in [Che04, ACSYD05]. In both cases the mesh energy is the *L1* error resulting from the interpolation of the function $f(x) = x^2$ by a piecewise linear function. In the case of the Lloyd smoother, the interpolation is linear in each Voronoi cell of the set of mesh vertices. In the case of the odt-smoother, the interpolation is linear in each cell of the Delaunay triangulation of the mesh vertices, hence the name odt which is an abbreviation for "optimal Delaunay triangulation".

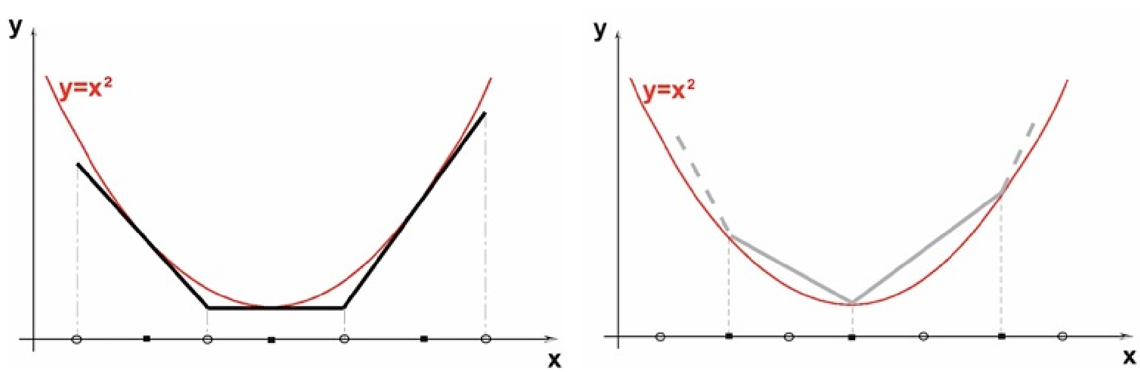


Figure 51.2: The one dimensional illustration of the mesh energy minimized by Lloyd (right) and ODT (left) smoothers.

The Lloyd optimizer is known to be blind to the occurrence of slivers in the mesh while the odt-smoother tends to chase them out. Both of them are global optimizers, meaning that they try to improve the whole mesh rather than focusing on the worst elements. However, both are empirically known to be very efficient as a preliminary step of optimization, as they tend to enhance the efficiency of the perturber and/or exuder applied next, see Figure 51.3

The perturber and the exuder focus on improving the worst mesh elements. The perturber [TSA09] improves the meshes by local changes in the vertices positions aiming to make sliver disappear. The exuder [CDE+00] chases the remaining slivers by re-weighting mesh vertices with optimal weights.

Each optimization process can be activated or not, and tuned according to the user requirements and the available time. By default, only the perturber and the exuder are activated.

Optimization processes are designed to improve mesh quality. However, beware that such an improvement is obtained by perturbing mesh vertices and modifying the mesh connectivity which has an impact on the strict compliance to the refinement criteria. Though a strict compliance to mesh criteria is granted at the end of the Delaunay refinement, this may no longer be true after some optimization processes. Also beware that the default behavior does involve some optimization processes.

51.2 Interface

The global functions

A 3D mesh generation process is launched through a call to one of the two following functions:

template <class C3T3, class MeshDomain_3, class MeshCriteria>
C3T3 make_mesh_3(MeshDomain_3 domain,

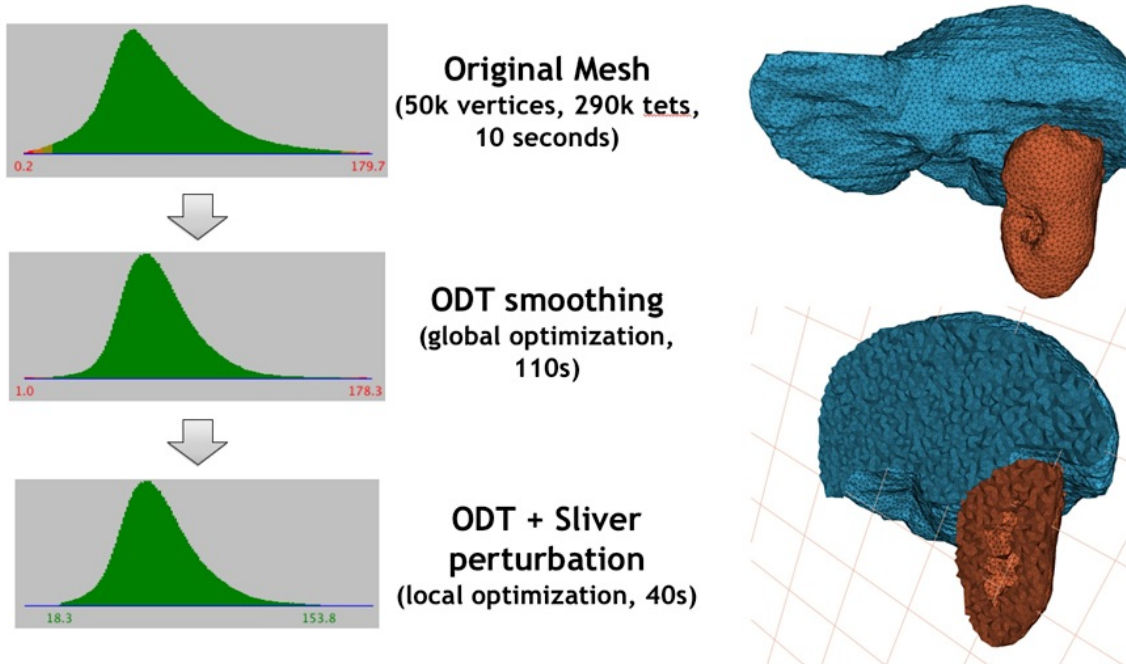


Figure 51.3: Compared effect of a global optimizer and the perturber. The left part shows the distribution of dihedral angles of mesh cells, right after Delaunay refinement (top), after some ODT smoothing (middle) and after perturbation (bottom). The numbers under the histograms give the measure in degrees of the smallest and biggest dihedral angles in the mesh.

MeshCriteria criteria,
parameters::internal::Features_options features = parameters::features(domain),
parameters::internal::Lloyd_options lloyd = parameters::no_lloyd(),
parameters::internal::Odt_options odt = parameters::no_odt(),
parameters::internal::Perturb_options perturb = parameters::perturb(),
parameters::internal::Exude_options exude = parameters::exude())

template <class C3T3, class MeshDomain_3, class MeshCriteria>
void refine_mesh_3(C3T3& c3t3,
MeshDomain_3 domain,
MeshCriteria criteria,
parameters::internal::Lloyd_options lloyd = parameters::no_lloyd(),
parameters::internal::Odt_options odt = parameters::no_odt(),
parameters::internal::Perturb_options perturb = parameters::perturb(),
parameters::internal::Exude_options exude = parameters::exude())

The function *make_mesh_3* generates from scratch a mesh of the input domain, while the function *refine_mesh_3* refines an existing mesh of the input domain. Note that as the protection of 0- and 1-dimensional features does not rely on Delaunay refinement, the function *refine_mesh_3* has no parameter to preserve features.

The data structure

The template parameter *C3T3* is required to be a model of the concept *MeshComplex_3InTriangulation_3*, a data structure devised to represent a three dimensional complex embedded in a 3D triangulation. In both functions, an instance of type *C3T3* is used to maintain the current approximating simplicial mesh and to represent the final 3D mesh at the end of the procedure. The type *C3T3* is required to provide a nested type *C3T3::Triangulation_3* for the 3D triangulation embedding the mesh. This triangulation is required to be a *CGAL::Regular_triangulation_3*. with vertex and cell base classes that are respectively models of the concepts *MeshVertexBase_3* and *MeshCellBase_3*.

The domain oracle and the features parameter

The template parameter *MeshDomain_3* is required to be a model of the concept *MeshDomain_3*. The argument *domain* of type *MeshDomain_3* is the sole link through which the domain to be discretized is known by the mesh generation algorithm.

This concept provides, among others, member functions to test whether or not a query segment intersects boundary surfaces, and to compute an intersection point in the affirmative. The *MeshDomain_3* concept adds member functions which given a query point tell whether the point lies inside or outside the domain and in which subdomain the point lies if inside.

If the domain description includes 0 and 1-dimensional features that have to be accurately represented in the final mesh, the template parameter *MeshDomain_3* is required to be of a model of the refined concept *MeshDomainWithFeatures_3*. Mainly the concept *MeshDomainWithFeatures_3* provides the incidence graph of 0, 1 and 2-dimensional features, and a member function to construct sample points on curve segments.

Using the parameter of type *Features*, the user whose domain is a model of *MeshDomainWithFeatures_3* can choose to have the corners and curve segments of the domain represented in the mesh or not. The type *Features* of this parameter is an internal undescribed type. The library provides functions to construct appropriate values of that type.

- *parameters::features(domain)* sets *features* according to the domain, i.e. 0 and 1-dimensional features are taken into account if *domain* is a *MeshDomainWithFeatures_3*
- *parameters::no_features()* prevents the representation of 0 and 1-dimensional features in the mesh. This is useful to get a smooth and rough approximation of a domain with features.

The meshing criteria

The template parameter *MeshCriteria* must be a model of the concept *MeshCriteria_3*, or a model of the refined concept *MeshCriteriaWithFeatures_3* if the domain has exposed features. The argument of type *MeshCriteria* passed to the mesh generator specifies the size and shape requirements for the tetrahedra in the mesh and for the triangles in the boundary surface mesh. These criteria condition the rules that drive the refinement process. At the end of the refinement process, mesh elements satisfy the criteria. This may not be strictly true anymore after the optimization phase, but this last phase is devised to only improve the mesh quality.

The criteria for surface facets are governed by the four following parameters:

- *facet_angle*. This parameter controls the shape of surface facets. Actually, it is a lower bound for the angle (in degree) of surface facets. When boundary surfaces are smooth, the termination of the meshing process is guaranteed if the angular bound is at most 30 degrees.

- *facet_size*. This parameter controls the size of surface facets. Actually, each surface facet has a surface Delaunay ball which is a ball circumscribing the surface facet and centered on the surface patch. The parameter *facet_size* is either a constant or a spatially variable scalar field, providing an upper bound for the radii of surface Delaunay balls.
- *facet_distance*. This parameter controls the approximation error of boundary and subdivision surfaces. Actually, it is either a constant or a spatially variable scalar field. It provides an upper bound for the distance between the circumcenter of a surface facet and the center of a surface Delaunay ball of this facet.
- *facet_topology*. This parameters controls the set of topological constraints which have to be verified by each surface facet. By default, each vertex of a surface facet has to be located on a surface patch, on a curve segment, or on a corner. It can also be set to check whether the three vertices of a surface facet belongs to the same surface patch. This has to be done cautiously, as such a criteria needs that each surface patches intersection is an input 1-dimensional feature.

The criteria for mesh cells are governed by two parameters:

- *cell_radius_edge_ratio*. This parameter controls the shape of mesh cells (but can't filter slivers, as we discussed earlier). Actually, it is an upper bound for the ratio between the circumradius of a mesh tetrahedron and its shortest edge. There is a theoretical bound for this parameter: the Delaunay refinement process is guaranteed to terminate for values of *cell_radius_edge_ratio* bigger than 2.
- *cell_size*. This parameter controls the size of mesh tetrahedra. It is either a scalar or a spatially variable scalar field. It provides an upper bound on the circumradii of the mesh tetrahedra.

Figure 51.4 shows how the mesh generation process behaves with respect to these parameters.

If the domain has 1-dimensional exposed features, the *criteria* includes a sizing field to guide the sampling of 1-dimensional features with protecting balls centers.

- *edge_size*. This constant or variable scalar field is used as an upper bound for the distance between two protecting ball centers that are consecutive on a 1-feature.

The optimization parameters

The four additional parameters are optimization parameters. They control which optimization processes are performed and allow the user to tune the parameters of the activated optimization processes. These parameters have internal types which are not described but the library provides global functions to generate appropriate values of these types:

- *parameters::lloyd()* and *parameters::no_lloyd()* activate and deactivate the Lloyd smoother.
- *parameters::odt()* and *parameters::no_odt()* activate and deactivate the odt-smoother.
- *parameters::perturb()* and *parameters::no_perturb()* activate and deactivate the perturber.
- *parameters::exude()* and *parameters::no_exude()* activate and deactivate the exuder.

These parameters are optional and can be passed in any order. If one parameter is not passed the default value is used. By default, only the perturber and the exuder are activated. Note that whatever may be the optimization

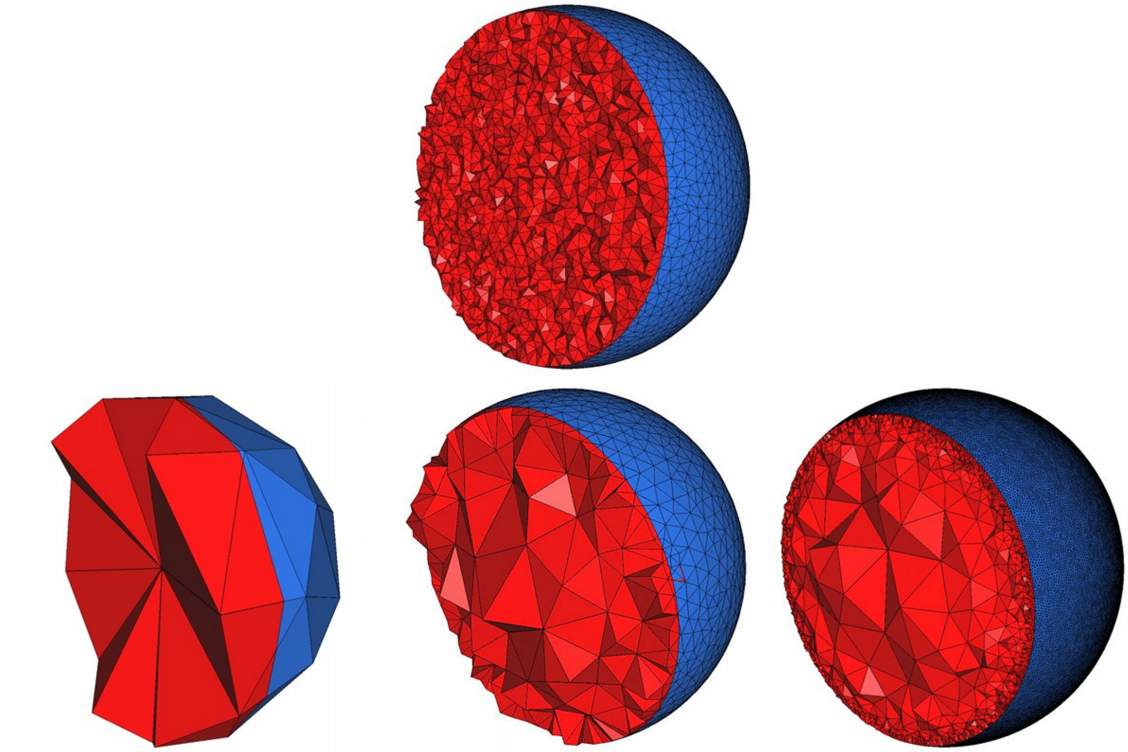


Figure 51.4: Top : the mesh is obtained using the parameters $(25, 0.15, 0.05)$ for the angular bound, radius bound and distance bound of surface facets and $(4, 0.2)$ for the radius-edge bound and radius bound of mesh cells. The result is a uniform mesh which contains tetrahedra of about the same size. Bottom left : the mesh is obtained by relaxing the *size bound* of tetrahedra and facets. The result is a small coarse mesh. Bottom middle : the mesh is obtained from the previous one by tightening the *distance bound* of surface facets to 0.01. The result is then a graded 3D mesh with a dense surface mesh achieving a precise approximation. Bottom right : the mesh is obtained from the previous one by fixing *radius bound* of surface facets to 0.01. The surface mesh is then denser to achieve the size bound.

processes activated by *make_mesh_3* or *refine_mesh_3*, they are always launched in the order that is a suborder of the following: *odt smoother*, *Lloyd smoother*, *perturber* and *exuder*.

The package also provides four global functions to launch each optimization process independently. These functions are useful for advanced experimentation on the efficiency of each optimization method. Note however that the exuder adds on mesh vertices weights that are conditioned by vertices positions. Therefore an exudation process should never be run before a smoother or a perturber. For a maximum efficiency, whatever may be the optimization processes activated, they should be launched in the order that is a suborder of the following: *odt-smoother*, *Lloyd-smoother*, *perturber*, *exuder*.

template< class C3T3, class MeshDomain_3 >
Mesh_optimization_return_code lloyd_optimize_mesh_3(C3T3& c3t3, MeshDomain_3 domain)

template< class C3T3, class MeshDomain_3 >
Mesh_optimization_return_code odt_optimize_mesh_3(C3T3& c3t3, MeshDomain_3 domain)

template< class C3T3, class MeshDomain_3 >
Mesh_optimization_return_code perturb_mesh_3(C3T3& c3t3, MeshDomain_3 domain)

template< class C3T3 >
Mesh_optimization_return_code *exude_mesh_3(C3T3& c3t3)*

Note that the global functions activating the optimization processes or launching those processes have themselves parameters (see details in reference pages) to tune the optimization process.

51.3 Examples

51.3.1 3D Domains Bounded by Isosurfaces

The following code produces a 3D mesh for a domain whose boundary surface is an isosurface defined by an implicit function. Figure 51.5 shows a cut view of the resulting mesh.

Note the use of named parameters (from Boost library) in the constructor of the *Mesh_criteria* instance.

```
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>

#include <CGAL/Mesh_triangulation_3.h>
#include <CGAL/Mesh_complex_3_in_triangulation_3.h>
#include <CGAL/Mesh_criteria_3.h>

#include <CGAL/Implicit_mesh_domain_3.h>
#include <CGAL/make_mesh_3.h>

// Domain
typedef CGAL::Exact_predicates_inexact_constructions_kernel K;
typedef K::FT FT;
typedef K::Point_3 Point;
typedef FT (Function)(const Point&);
typedef CGAL::Implicit_mesh_domain_3<Function,K> Mesh_domain;

// Triangulation
typedef CGAL::Mesh_triangulation_3<Mesh_domain>::type Tr;
typedef CGAL::Mesh_complex_3_in_triangulation_3<Tr> C3t3;

// Criteria
typedef CGAL::Mesh_criteria_3<Tr> Mesh_criteria;

// To avoid verbose function and named parameters call
using namespace CGAL::parameters;

// Function
FT sphere_function (const Point& p)
{ return CGAL::squared_distance(p, Point(CGAL::ORIGIN))-1; }

int main()
{
  // Domain (Warning: Sphere_3 constructor uses squared radius !)
  Mesh_domain domain(sphere_function, K::Sphere_3(CGAL::ORIGIN, 2.));

  // Mesh criteria
```

```
  Mesh_criteria criteria(facet_angle=30, facet_size=0.1,
   facet_distance=0.025,
                              cell_radius_edge_ratio=2, cell_size=0.1);

  // Mesh generation
  C3t3 c3t3 = CGAL::make_mesh_3<C3t3>(domain, criteria);

  // Output
  std::ofstream medit_file("out.mesh");
  c3t3.output_to_medit(medit_file);

  return 0;
}
```

File: examples/Mesh_3/mesh_implicit_sphere.cpp

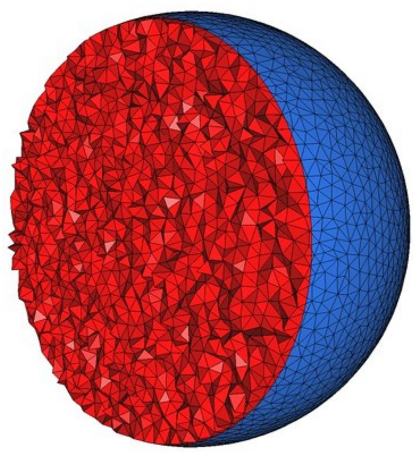


Figure 51.5: Cut view of a 3D mesh produced from an implicit domain

51.3.2 3D Polyhedral Domains

The following code produces a 3D mesh for a domain defined by polyhedral surfaces. Figure 51.6 shows the resulting mesh.

```
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>

#include <CGAL/Mesh_triangulation_3.h>
#include <CGAL/Mesh_complex_3_in_triangulation_3.h>
#include <CGAL/Mesh_criteria_3.h>

#include <CGAL/Polyhedral_mesh_domain_3.h>
#include <CGAL/make_mesh_3.h>
#include <CGAL/refine_mesh_3.h>

// IO
#include <CGAL/IO/Polyhedron_iostream.h>
```

```
// Domain
typedef CGAL::Exact_predicates_inexact_constructions_kernel K;
typedef CGAL::Polyhedron_3<K> Polyhedron;
typedef CGAL::Polyhedral_mesh_domain_3<Polyhedron, K> Mesh_domain;

// Triangulation
typedef CGAL::Mesh_triangulation_3<Mesh_domain>::type Tr;
typedef CGAL::Mesh_complex_3_in_triangulation_3<Tr> C3t3;

// Criteria
typedef CGAL::Mesh_criteria_3<Tr> Mesh_criteria;

// To avoid verbose function and named parameters call
using namespace CGAL::parameters;

int main()
{
  // Create input polyhedron
  Polyhedron polyhedron;
  std::ifstream input("data/elephant.off");
  input >> polyhedron;

  // Create domain
  Mesh_domain domain(polyhedron);

  // Mesh criteria (no cell_size set)
  Mesh_criteria criteria(facet_angle=25, facet_size=0.15,
   facet_distance=0.008,
                         cell_radius_edge_ratio=3);

  // Mesh generation
  C3t3 c3t3 = CGAL::make_mesh_3<C3t3>(domain, criteria, no_perturb(),
   no_exude());

  // Output
  std::ofstream medit_file("out_1.mesh");
  c3t3.output_to_medit(medit_file);
  medit_file.close();

  // Set tetrahedron size (keep cell_radius_edge_ratio), ignore facets
  Mesh_criteria new_criteria(cell_radius_edge_ratio=3, cell_size=0.03);

  // Mesh refinement
  CGAL::refine_mesh_3(c3t3, domain, new_criteria);

  // Output
  medit_file.open("out_2.mesh");
  c3t3.output_to_medit(medit_file);

  return 0;
}
```

File: examples/Mesh_3/mesh_polyhedral_domain.cpp

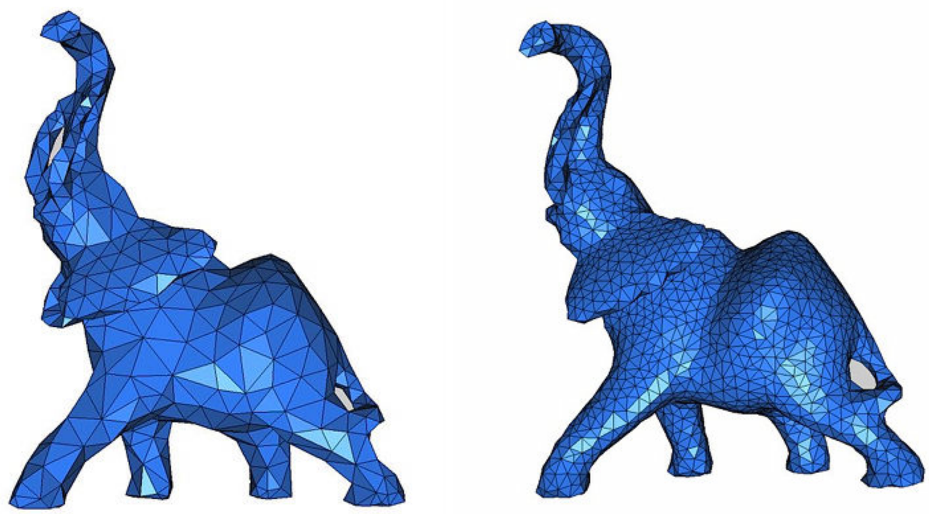


Figure 51.6: View of 3D meshes produced from a polyhedral domain. (i) is a view of file out_1.mesh and (ii) is a view of file out_2.mesh. Code from subsection 51.3.2 generates these files.

51.3.3 Domains From Segmented 3D Images

The following code produces a 3D mesh from a 3D image. The image is a segmented medical image in which each voxel is associated a label in accordance with the tissue the voxel belongs to. The domain is therefore a multi-domain where each subdomain corresponds to a specific tissue.

In the following example, the image is read from the file *liner.inr.gz* which is encoded in the format of the library Inrimage *http://inrimage.gforge.inria.fr/*. The resulting mesh is shown in Figure 51.7.

```
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>

#include <CGAL/Mesh_triangulation_3.h>
#include <CGAL/Mesh_complex_3_in_triangulation_3.h>
#include <CGAL/Mesh_criteria_3.h>

#include <CGAL/Labeled_image_mesh_domain_3.h>
#include <CGAL/make_mesh_3.h>
#include <CGAL/Image_3.h>

// Domain
typedef CGAL::Exact_predicates_inexact_constructions_kernel K;
typedef CGAL::Labeled_image_mesh_domain_3<CGAL::Image_3,K> Mesh_domain;

// Triangulation
typedef CGAL::Mesh_triangulation_3<Mesh_domain>::type Tr;
typedef CGAL::Mesh_complex_3_in_triangulation_3<Tr> C3t3;

// Criteria
typedef CGAL::Mesh_criteria_3<Tr> Mesh_criteria;
```

```
// To avoid verbose function and named parameters call
using namespace CGAL::parameters;

int main()
{
  // Loads image
  CGAL::Image_3 image;
  image.read("data/liver.inr.gz");

  // Domain
  Mesh_domain domain(image);

  // Mesh criteria
  Mesh_criteria criteria(facet_angle=30, facet_size=6, facet_distance=4,
                         cell_radius_edge_ratio=3, cell_size=8);

  // Meshing
  C3t3 c3t3 = CGAL::make_mesh_3<C3t3>(domain, criteria);

  // Output
  std::ofstream medit_file("out.mesh");
  c3t3.output_to_medit(medit_file);

  return 0;
}
```

File: examples/Mesh_3/mesh_3D_image.cpp

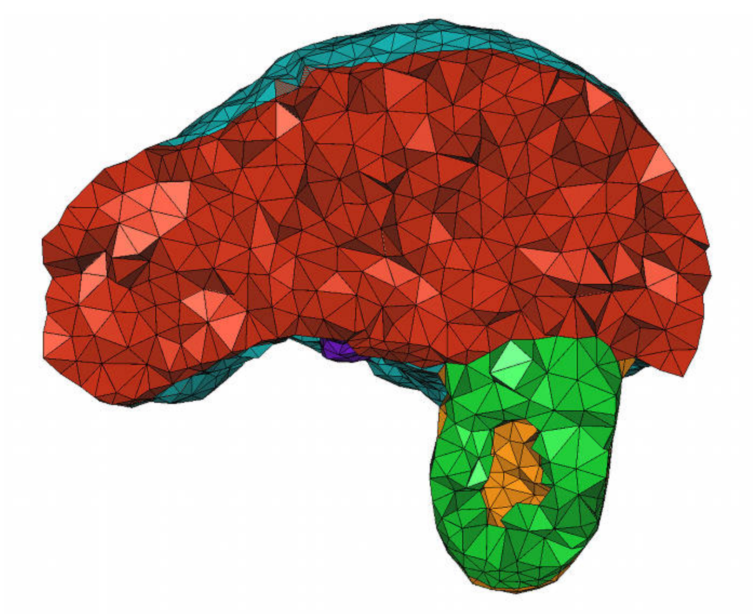


Figure 51.7: Cut view of a 3D mesh produced from a segmented liver image. Code from subsection 51.3.3 generates this file.

51.3.4 Using Variable Sizing Field

Sizing field as an analytical function

The following example shows how to use an analytical function as sizing field.

```
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>

#include <CGAL/Mesh_triangulation_3.h>
#include <CGAL/Mesh_complex_3_in_triangulation_3.h>
#include <CGAL/Mesh_criteria_3.h>

#include <CGAL/Implicit_mesh_domain_3.h>
#include <CGAL/make_mesh_3.h>

// Domain
typedef CGAL::Exact_predicates_inexact_constructions_kernel K;
typedef K::FT FT;
typedef K::Point_3 Point;
typedef FT (Function)(const Point&);
typedef CGAL::Implicit_mesh_domain_3<Function,K> Mesh_domain;

// Triangulation
typedef CGAL::Mesh_triangulation_3<Mesh_domain>::type Tr;
typedef CGAL::Mesh_complex_3_in_triangulation_3<Tr> C3t3;

// Criteria
typedef CGAL::Mesh_criteria_3<Tr> Mesh_criteria;

// Sizing field
struct Spherical_sizing_field
{
  typedef ::FT FT;
  typedef Point Point_3;
  typedef Mesh_domain::Index Index;

  FT operator()(const Point_3& p, const int, const Index&) const
  {
    FT sq_d_to_origin = CGAL::squared_distance(p, Point(CGAL::ORIGIN));
    return CGAL::abs( CGAL::sqrt(sq_d_to_origin)-0.5 ) / 5. + 0.025;
  }
};

// To avoid verbose function and named parameters call
using namespace CGAL::parameters;

// Function
FT sphere_function (const Point& p)
{ return CGAL::squared_distance(p, Point(CGAL::ORIGIN))-1; }

int main()
{
  // Domain (Warning: Sphere_3 constructor uses squared radius !)
  Mesh_domain domain(sphere_function, K::Sphere_3(CGAL::ORIGIN, 2.));
```

```
  // Mesh criteria
  Spherical_sizing_field size;
  Mesh_criteria criteria(facet_angle=30, facet_size=0.1,
   facet_distance=0.025,
                         cell_radius_edge_ratio=2, cell_size=size);

  // Mesh generation
  C3t3 c3t3 = CGAL::make_mesh_3<C3t3>(domain, criteria, no_exude(),
   no_perturb());

  // Output
  std::ofstream medit_file("out.mesh");
  c3t3.output_to_medit(medit_file);

  return 0;
}
```

File: examples/Mesh_3/mesh_implicit_sphere_variable_size.cpp

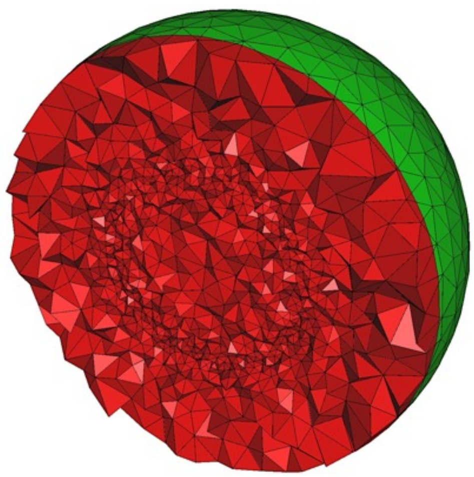


Figure 51.8: Cut view of a 3D mesh produced from an implicit sphere with non-constant sizing field. Code from subsection 51.3.4 generates this file.

Different Sizing Field for Different Subdomains

The following example shows how to use different size for different organs in a 3D medical image.

```
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>

#include <CGAL/Mesh_triangulation_3.h>
#include <CGAL/Mesh_complex_3_in_triangulation_3.h>
#include <CGAL/Mesh_criteria_3.h>
#include <CGAL/Mesh_constant_domain_field_3.h>
```

```
#include <CGAL/Labeled_image_mesh_domain_3.h>
#include <CGAL/make_mesh_3.h>
#include <CGAL/Image_3.h>

// Domain
typedef CGAL::Exact_predicates_inexact_constructions_kernel K;
typedef CGAL::Labeled_image_mesh_domain_3<CGAL::Image_3,K> Mesh_domain;

// Triangulation
typedef CGAL::Mesh_triangulation_3<Mesh_domain>::type Tr;
typedef CGAL::Mesh_complex_3_in_triangulation_3<Tr> C3t3;

// Criteria
typedef CGAL::Mesh_criteria_3<Tr> Mesh_criteria;
typedef CGAL::Mesh_constant_domain_field_3<Mesh_domain::R,
                                           Mesh_domain::Index>
   Sizing_field;

// To avoid verbose function and named parameters call
using namespace CGAL::parameters;

int main()
{
  // Loads image
  CGAL::Image_3 image;
  image.read("data/liver.inr.gz");

  // Domain
  Mesh_domain domain(image);

  // Sizing field: set global size to 8 and kidney size (label 127) to 3
  double kidney_size = 3.;
  int volume_dimension = 3;
  Sizing_field size(8);
  size.set_size(kidney_size, volume_dimension,
                domain.index_from_subdomain_index(127));

  // Mesh criteria
  Mesh_criteria criteria(facet_angle=30, facet_size=6, facet_distance=2,
                         cell_radius_edge_ratio=3, cell_size=size);

  // Meshing
  C3t3 c3t3 = CGAL::make_mesh_3<C3t3>(domain, criteria);

  // Output
  std::ofstream medit_file("out.mesh");
  c3t3.output_to_medit(medit_file);

  return 0;
}
```

File: examples/Mesh_3/mesh_3D_image_variable_size.cpp

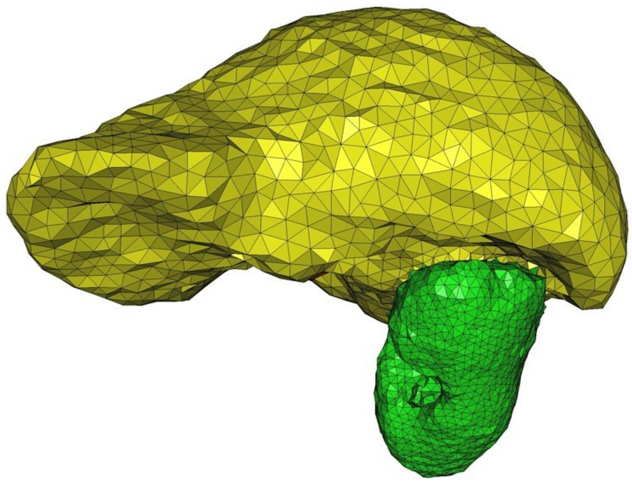


Figure 51.9: View of a 3D mesh produced from a 3D image with different size for different organs. Code from subsection 51.3.4 generates this file.

51.3.5 Meshing Domains with Sharp Features

3D polyhedral domain with edges

The following example shows how to generate a mesh from a polyhedral surface. The output mesh conforms to the sharp features of the input surface.

```
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>

#include <CGAL/Mesh_triangulation_3.h>
#include <CGAL/Mesh_complex_3_in_triangulation_3.h>
#include <CGAL/Mesh_criteria_3.h>

#include <CGAL/Polyhedral_mesh_domain_with_features_3.h>
#include <CGAL/make_mesh_3.h>

// Domain
typedef CGAL::Exact_predicates_inexact_constructions_kernel K;
typedef CGAL::Polyhedral_mesh_domain_with_features_3<K> Mesh_domain;

// Triangulation
typedef CGAL::Mesh_triangulation_3<Mesh_domain>::type Tr;
typedef CGAL::Mesh_complex_3_in_triangulation_3<
  Tr,Mesh_domain::Corner_index,Mesh_domain::Curve_segment_index> C3t3;

// Criteria
typedef CGAL::Mesh_criteria_3<Tr> Mesh_criteria;

// To avoid verbose function and named parameters call
using namespace CGAL::parameters;

int main()
{
```

```
  // Create domain
  Mesh_domain domain("data/fandisk.off");

  // Get sharp features
  domain.detect_features();

  // Mesh criteria
  Mesh_criteria criteria(edge_size = 0.025,
                         facet_angle = 25, facet_size = 0.05,
   facet_distance = 0.005,
                         cell_radius_edge_ratio = 3, cell_size = 0.05);

  // Mesh generation
  C3t3 c3t3 = CGAL::make_mesh_3<C3t3>(domain, criteria);

  // Output
  std::ofstream medit_file("out.mesh");
  c3t3.output_to_medit(medit_file);
}
```

File: examples/Mesh_3/mesh_polyhedral_domain_with_features.cpp

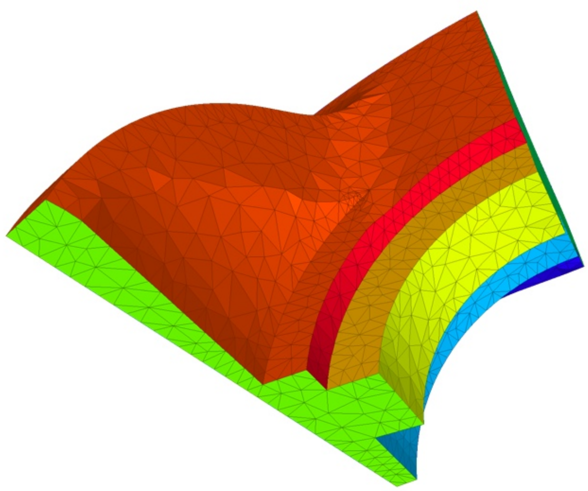


Figure 51.10: View of a 3D mesh with sharp features. Code from subsection 51.3.5 generates this mesh.

Implicit domain with 1D features

The following example shows how to generate a mesh from an implicit domain. We add by hand the intersection of the spheres as a sharp feature.

```
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>

#include <CGAL/Mesh_triangulation_3.h>
#include <CGAL/Mesh_complex_3_in_triangulation_3.h>
#include <CGAL/Mesh_criteria_3.h>
```

```
#include <CGAL/Implicit_mesh_domain_3.h>
#include <CGAL/Mesh_domain_with_polyline_features_3.h>
#include <CGAL/make_mesh_3.h>

// Kernel
typedef CGAL::Exact_predicates_inexact_constructions_kernel K;

// Domain
typedef K::FT FT;
typedef K::Point_3 Point;
typedef FT (Function)(const Point&);
typedef CGAL::Mesh_domain_with_polyline_features_3<
  CGAL::Implicit_mesh_domain_3<Function,K> >              Mesh_domain;

// Polyline
typedef std::vector<Point>        Polyline_3;
typedef std::list<Polyline_3>       Polylines;

// Triangulation
typedef CGAL::Mesh_triangulation_3<Mesh_domain>::type Tr;
typedef CGAL::Mesh_complex_3_in_triangulation_3<
  Tr,Mesh_domain::Corner_index,Mesh_domain::Curve_segment_index> C3t3;

// Criteria
typedef CGAL::Mesh_criteria_3<Tr> Mesh_criteria;

// To avoid verbose function and named parameters call
using namespace CGAL::parameters;

// Function
FT sphere_function1 (const Point& p)
{ return CGAL::squared_distance(p, Point(CGAL::ORIGIN))-2; }

FT sphere_function2 (const Point& p)
{ return CGAL::squared_distance(p, Point(2, 0, 0))-2; }

FT sphere_function (const Point& p)
{
  if(sphere_function1(p) < 0 || sphere_function2(p) < 0)
    return -1;
  else
    return 1;
}

#include <cmath>

int main()
{
  // Domain (Warning: Sphere_3 constructor uses squared radius !)
  Mesh_domain domain(sphere_function, K::Sphere_3(Point(1, 0, 0), 6.));

  // Mesh criteria
  Mesh_criteria criteria(edge_size = 0.15,
```

```
                             facet_angle = 25, facet_size = 0.15,
                             cell_radius_edge_ratio = 2, cell_size = 0.15);

  // Create edge that we want to preserve
  Polylines polylines (1);
  Polyline_3& polyline = polylines.front();

  for(int i = 0; i < 360; ++i)
  {
    Point p (1, std::cos(i*CGAL_PI/180), std::sin(i*CGAL_PI/180));
    polyline.push_back(p);
  }
  polyline.push_back(polyline.front()); // close the line

  // Insert edge in domain
  domain.add_features(polylines.begin(), polylines.end());

  // Mesh generation without feature preservation
  C3t3 c3t3 = CGAL::make_mesh_3<C3t3>(domain, criteria,
                                      CGAL::parameters::no_features());

  std::ofstream medit_file("out-no-protection.mesh");
  c3t3.output_to_medit(medit_file);
  medit_file.close();
  c3t3.clear();

  // Mesh generation with feature preservation
  c3t3 = CGAL::make_mesh_3<C3t3>(domain, criteria);

  // Output
  medit_file.open("out-with-protection.mesh");
  c3t3.output_to_medit(medit_file);
  medit_file.close();

  return 0;
}
```

File: examples/Mesh_3/mesh_two_implicit_spheres_with_balls.cpp

51.3.6 Tuning Mesh Optimization

In the previous examples, the mesh generation is launched through a call *make_mesh_3(domain,criteria)* with a minimal number of parameters. In such cases, the default optimization strategy is applied: after the Delaunay refinement process two optimization steps are performed, a perturbation and a sliver exudation. The following examples show how to disable default optimization steps and how to tune the parameters of optimization steps.

Disabling exudation and tuning perturbation

In this first example, we show how to disable the exudation step. The optimization phase after the refinement includes only a perturbation phase which is launched with no time bound and an objective of 10 degrees for the

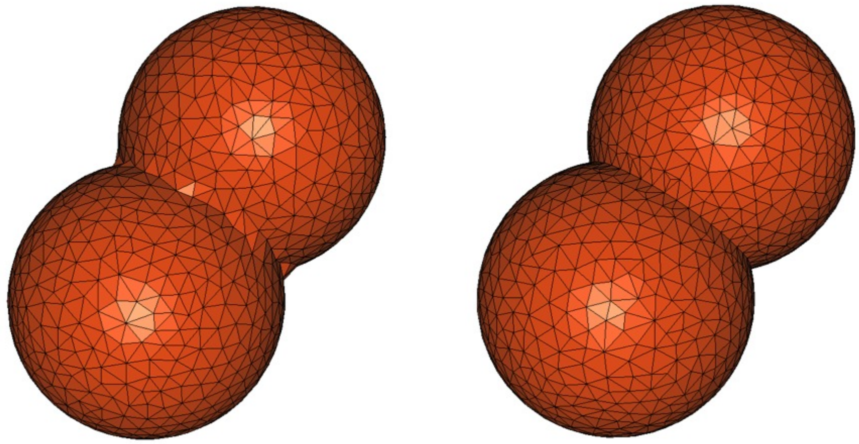


Figure 51.11: View of a 3D mesh with sharp features generated from two intersected implicit spheres. On the left, one can see the mesh without feature preservation, and on the right the mesh with feature preservation.

minimum dihedral angle of the mesh. The example shows two ways of achieving the same result. The first way issues a single call to *make_mesh_3* with the required optimization process activated and tuned. In the second way, *make_mesh_3* is first called without any optimization process and the resulting mesh is next optimized through a call to *perturb_mesh_3* with tuned parameters.

```
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>

#include <CGAL/Mesh_triangulation_3.h>
#include <CGAL/Mesh_complex_3_in_triangulation_3.h>
#include <CGAL/Mesh_criteria_3.h>

#include <CGAL/Labeled_image_mesh_domain_3.h>
#include <CGAL/make_mesh_3.h>
#include <CGAL/Image_3.h>

// Domain
typedef CGAL::Exact_predicates_inexact_constructions_kernel K;
typedef CGAL::Labeled_image_mesh_domain_3<CGAL::Image_3,K> Mesh_domain;

// Triangulation
typedef CGAL::Mesh_triangulation_3<Mesh_domain>::type Tr;
typedef CGAL::Mesh_complex_3_in_triangulation_3<Tr> C3t3;

// Mesh Criteria
typedef CGAL::Mesh_criteria_3<Tr> Mesh_criteria;

// To avoid verbose function and named parameters call
using namespace CGAL::parameters;

int main()
{
  // Domain
  CGAL::Image_3 image;
  image.read("data/liver.inr.gz");
  Mesh_domain domain(image);
```

```
  // Mesh criteria
  Mesh_criteria criteria(facet_angle=30, facet_size=5,
    facet_distance=1.5,
                         cell_radius_edge_ratio=2, cell_size=7);

  // Mesh generation and optimization in one call (sliver_bound is the
  // targeted dihedral angle in degree)
  C3t3 c3t3 = CGAL::make_mesh_3<C3t3>(domain, criteria,
                                      no_exude(),
                                      perturb(sliver_bound=10,
    time_limit=15));

  // Mesh generation and optimization in several call
  C3t3 c3t3_bis = CGAL::make_mesh_3<C3t3>(domain, criteria,
                                          no_perturb(), no_exude());

  CGAL::perturb_mesh_3(c3t3_bis, domain, time_limit=15);

  // Output
  std::ofstream medit_file("out.mesh");
  c3t3.output_to_medit(medit_file);

  std::ofstream medit_file_bis("out_bis.mesh");
  c3t3_bis.output_to_medit(medit_file_bis);

  return 0;
}
```

File: examples/Mesh_3/mesh_optimization_example.cpp

Using Lloyd global optimization

In this second example, we show how to call the Lloyd optimization on the mesh, followed by a call to exudation. We set a time bound of 30s for the Lloyd optimization. We set a time bound of 10s and a sliver bound of 10 degrees for the exuder.

```
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>

#include <CGAL/Mesh_triangulation_3.h>
#include <CGAL/Mesh_complex_3_in_triangulation_3.h>
#include <CGAL/Mesh_criteria_3.h>

#include <CGAL/Labeled_image_mesh_domain_3.h>
#include <CGAL/make_mesh_3.h>
#include <CGAL/Image_3.h>

// Domain
typedef CGAL::Exact_predicates_inexact_constructions_kernel K;
typedef CGAL::Labeled_image_mesh_domain_3<CGAL::Image_3,K> Mesh_domain;

// Triangulation
```

```
typedef CGAL::Mesh_triangulation_3<Mesh_domain>::type Tr;
typedef CGAL::Mesh_complex_3_in_triangulation_3<Tr> C3t3;

// Mesh Criteria
typedef CGAL::Mesh_criteria_3<Tr> Mesh_criteria;

// To avoid verbose function and named parameters call
using namespace CGAL::parameters;

int main()
{
  // Domain
  CGAL::Image_3 image;
  image.read("data/liver.inr.gz");
  Mesh_domain domain(image);

  // Mesh criteria
  Mesh_criteria criteria(facet_angle=30, facet_distance=1.2,
                         cell_radius_edge_ratio=2);

  // Mesh generation and optimization in one call
  C3t3 c3t3 = CGAL::make_mesh_3<C3t3>(domain, criteria,
                                      lloyd(time_limit=30),
                                      no_perturb(),
                                      exude(time_limit=10,
   sliver_bound=10));

  // Mesh generation and optimization in several call
  C3t3 c3t3_bis = CGAL::make_mesh_3<C3t3>(domain, criteria,
                                          no_perturb(), no_exude());

  CGAL::lloyd_optimize_mesh_3(c3t3_bis, domain, time_limit=30);
  CGAL::exude_mesh_3(c3t3_bis, sliver_bound=10, time_limit=10);

  // Output
  std::ofstream medit_file("out.mesh");
  c3t3.output_to_medit(medit_file);

  std::ofstream medit_file_bis("out_bis.mesh");
  c3t3_bis.output_to_medit(medit_file_bis);

  return 0;
}
```

File: examples/Mesh_3/mesh_optimization_lloyd_example.cpp

51.4 Performances

We provide here some benchmarks of the performances of the mesh generation engine. The machine used is a PC running Linux64 with two Intel Xeon CPU X5450 clocked at 3.00 GHz with 32GB of RAM. The program has been compiled with g++ v4.3.2 with the -O3 option. Note that our implementation does not take advantage of multi-core architectures.

Those benchmarks have been done using CGAL v3.8.

51.4.1 Delaunay refinement

We study the refinement part of the mesh generation engine in this section. We give the CPU time (measured by *CGAL::Timer*) using the 3 provided oracles. In all experiments, we produce well shaped elements: we set the facet angle bound and the radius edge bound to their theoretical limit (resp. 30 degrees and 2). We also use the same uniform sizing field for facets and cells.

Implicit function

We mesh an analytical sphere of radius 1.

Size bound	vertices nb	facets nb	tetrahedra nb	CPU Time (s)	vertices/second
0.2	499	488	2,299	0.0240	20,800
0.1	3,480	2,046	18,756	0.146	23,800
0.05	25,556	8,274	149,703	1.50	17,000
0.025	195,506	33,212	1,194,727	17.4	11,200
0.0125	1,528,636	134,810	9,547,772	179	8,530

Polyhedral domain

We mesh a volume bounded by a closed triangulated surface made of about 50,000 vertices and 100,000 triangles. Picture 51.12 shows the mesh obtained when size is set to 0.005.

Size bound	vertices nb	facets nb	tetrahedra nb	CPU Time (s)	vertices/second
0.04	423	717	1,332	0.488	866
0.02	2,638	3,414	10,957	2.64	998
0.01	18,168	15,576	90,338	13.9	1,310
0.005	129,442	64,645	722,018	66.7	1,940
0.0025	967,402	263,720	5,756,491	348	2,780

3D image

We mesh image number 2 from the 3D-IRCADb-01[1] public database. The size of this image is 512x512x172 voxels (about 45M voxels). The size of the voxels is 0.78mm x 0.78mm x 1.6mm. Picture 51.13 shows the mesh obtained for size set to 4.

Size bound (mm)	vertices nb	facets nb	tetrahedra nb	CPU Time (s)	vertices/second
16	3,898	4,099	20,692	0.344	11,300
8	34,117	27,792	199,864	3.09	11,000
4	206,566	86,180	1,253,694	22.4	9,230
2	1,546,196	329,758	9,617,278	199	7,780

[1] available at http://www.ircad.fr/softwares/3Dircadb/3Dircadb1/index.php

Figure 51.12: View of polyhedral mesh generation result (size = 0.005).

51.5 Design and Implementation History

Theoretical foundations

The CGAL mesh generation package implements a meshing engine based on the method of Delaunay refinement introduced by Chew [Che93] and Ruppert [Rup95] and pioneered in 3D by Shewchuk [She98b]. It uses the notion of restricted Delaunay triangulation to approximate 1-dimensional curved features and curved surface patches and rely on the work of Boissonnat and Oudot [BO05] and Oudot et al. [ORY05] to achieve accurate representation of boundary and subdividing surfaces in the mesh. The mechanism of protecting balls, used to ensure a fair representation of 1-dimensional features, if any, and the termination of the refinement process whatever may be the input geometry, in particular whatever small dihedral angles may form the boundary and subdivision surface patches, was pioneered by Cheng et al. [CDR07] and further experimented by Dey, Levine et al. [CDL07]. The optimization phase involves global optimization processes, a perturbation process and a sliver exudation process. The global optimizers are based on Lloyd smoothing [DFG99, DW02] and odt smoothing [Che04, ACSYD05], where odt means *optimal Delaunay triangulation*. The perturbation process is mainly based on the work of Tournois [Tou09] and Tournois et al. [TWAD09], while the exudation process is, the now famous, optimization by weighting described in Edelsbrunner et al. [CDE+00].

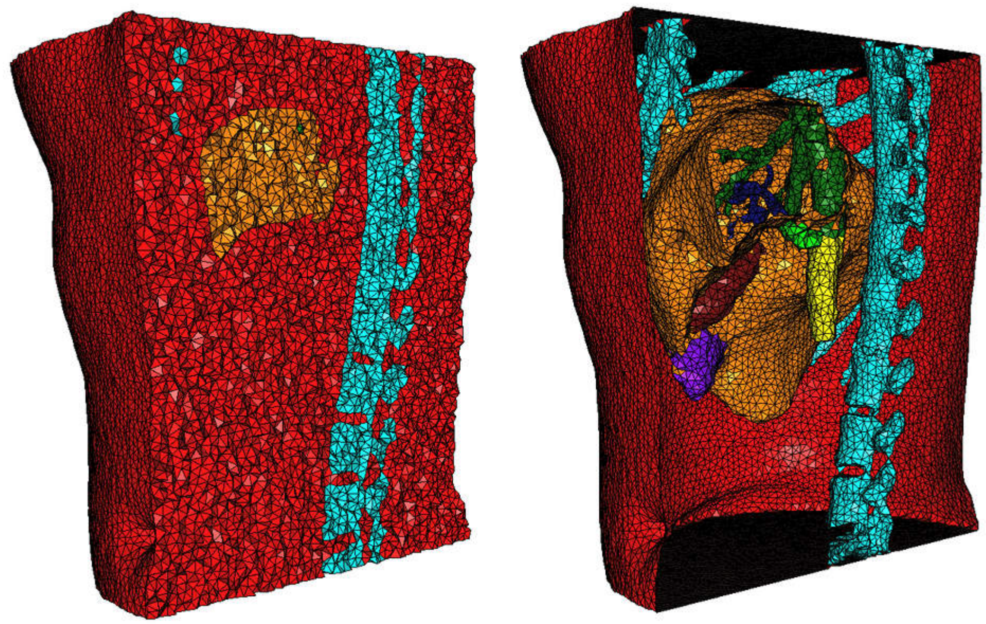


Figure 51.13: View of 3d image mesh generation result (size = 4).

Implementation history

Work on the package *Mesh_3* started during the PhD thesis of Laurent Rineau advised by Mariette Yvinec. A code prototype, together with a first version of design and specifications [RY07] came out of their collaboration.

From the beginning of 2009, most of the work has been performed by Stéphane Tayeb, in collaboration with Mariette Yvinec, Laurent Rineau, Pierre Alliez and Jane Tournois. First, Stéphane released the first public version of the package, implementing the specifications written by Laurent and Mariette.

The optimization processes are heavily based on the work of Jane Tournois and Pierre Alliez during the PhD of Jane advised by Pierre. The optimization phase was imported in the mesh generation package by Stéphane Tayeb and appeared first in release 3.6 of CGAL.

In collaboration with Laurent Rineau, Stéphane also added demos and examples. After some experiments on medical imaging data performed by Dobrina Boltcheva et al. [BYB09b, BYB09a], the handling of 1-dimensional features was worked out by Laurent Rineau, Stéphane Tayeb and Mariette Yvinec. It appeared first in the release 3.8 of CGAL.

3D Mesh Generation Reference Manual

Pierre Alliez, Laurent Rineau, Stéphane Tayeb, Jane Tournois, Mariette Yvinec

51.6 Classified Reference Pages

Concepts

Here are the main concepts of this package:

The following concepts are secondary concepts, related to the template parameters of some models of the main concepts:

Classes

The following classes are models of domain concepts and their associated classes:

Function Templates

CGAL::parameters Functions

Enumerations

51.7 Alphabetical List of Reference Pages

Concept

MeshComplex_3InTriangulation_3

Definition

The concept MeshComplex_3InTriangulation_3 describes a data structure to represent and maintain a 3D complex embedded in a 3D triangulation. More precisely, the concept MeshComplex_3InTriangulation_3 is a minimal version designed to represent 3D complexes that have only faces with dimension 2 and 3. Embedded 3D complexes with faces of dimension 0, 1, 2 and 3, are more conveniently represented by the refined concept *MeshComplexWithFeatures_3InTriangulation_3*.

The data structure includes a 3D triangulation which is itself a 3D complex. To distinguish the faces of the embedded 3D complex from the faces of the triangulation, we call the faces of the embedded complex respectively *subdomains*, for 3D faces and *surface patches*, for 2D faces, while the triangulations faces are called respectively cells, facets, edges and vertices.

Each subdomain of the embedded 3D complex is a union of triangulation cells. Cells that belong to some subdomain are said to belong to the embedded complex. Each surface patch is a union of triangulation facets. Triangulation facets that belong to some surface patch are called surface facets. The concept MeshComplex_3InTriangulation_3 handles the marking and retrieval of the cells of the triangulation belonging to the subdomains and of the facets of the triangulation belonging to the surface patches. The concept MeshComplex_3InTriangulation_3 also includes an index type for vertices of the triangulation and attaches an integer, called the *dimension* to each vertex. When used by the meshing algorithm, the index and the dimension of each vertex are used to store respectively the lowest dimensional face of the input complex including the vertex and the dimension of this face.

In the 3D mesh generator, the concept MeshComplex_3InTriangulation_3 is used when the domain to be meshed has no feature with dimension 0 and 1. Such a data structure is used internally by the mesh generator to maintain the current approximation of each subdomain and each boundary surface patch. The data structure encodes the final mesh at the end of the meshing process.

Types

MeshComplex_3InTriangulation_3:: Triangulation

Type of the embedding 3D triangulation. This type must be a 3D regular triangulation of CGAL when used for 3D mesh generation.

typedef Triangulation::Vertex_handle
Vertex_handle; — Type *Vertex_handle* type of the triangulation.

typedef Triangulation::Cell_handle
Cell_handle; — The *Cell_handle* type of the triangulation.

typedef Triangulation::Facet
Facet; — The *Facet* type of the triangulation.

typedef Triangulation::Edge
Edge; — The *Edge* type of the triangulation.

typedef Triangulation::size_type
size_type; — Size type (unsigned integral type).

MeshComplex_3InTriangulation_3:: Subdomain_index

A type for indices of subdomains. This type must match the type *MeshDomain_3::Subdomain_index* when the concept is used for mesh generation.

MeshComplex_3InTriangulation_3:: Surface_patch_index

A type for indices of surface patches. This type must match the type *MeshDomain_3::Surface_patch_index* when the concept is used for mesh generation.

MeshComplex_3InTriangulation_3:: Index — A type for indexing vertices that belong to some surface patches or subdomains. This type must match the type *MeshDomain_3::Index* when the concept is used for mesh generation.

MeshComplex_3InTriangulation_3:: Cells_in_complex_iterator

An iterator type to visit the cells of the triangulation belonging to the 3D complex.

MeshComplex_3InTriangulation_3:: Facets_in_complex_iterator

An iterator type to visit the surface facets.

Creation

MeshComplex_3InTriangulation_3 c3t3;	Builds an empty 3D complex.
MeshComplex_3InTriangulation_3 c3t3(rhs);	Copy constructor. Embedded triangulation is duplicated.

MeshComplex_3InTriangulation_3&

	c3t3 = rhs	Assignment operator. Embedded triangulation is duplicated, and the former triangulation of *c3t3* is deleted.
void	*c3t3.swap(& rhs)*	Swaps *c3t3* and *rhs*.

Access Functions

Triangulation *c3t3.triangulation() const*

Returns a const reference to the triangulation.

advanced

Non const access

The responsibility of keeping a valid C3T3 belongs to the user when using advanced operations allowing a direct manipulation of the triangulation.

Triangulation&	*c3t3.triangulation()*	Returns a reference to the triangulation.

advanced

Modifiers

void *c3t3.add_to_complex(Cell_handle c, Subdomain_index index)*

Sets the cell *c* as a cell of the subdomain *index*.

void *c3t3.add_to_complex(Facet f, Surface_patch_index index)*

Adds the facet *f* as a facet of the surface patch *index*.)

void *c3t3.add_to_complex(Cell_handle c, int i, Surface_patch_index index)*

Same as above with *f=(c,i)*.

void *c3t3.remove_from_complex(Cell_handle c)*

Removes cell *c* from the embedded complex.

void *c3t3.remove_from_complex(Facet f)*

Removes facet *f* from the embedded complex.

void *c3t3.remove_from_complex(Cell_handle c, int i)*

Same as above with *f=(c,i)*.

void *c3t3.set_dimension(Vertex_handle v, int dimension)*

Sets the "dimension" of vertex *v*. The dimension is an integer attached to the vertex. When the concept MeshComplex_3InTriangulation_3 is used for mesh generation this integer is used to store the dimension of the lowest dimensional face of the input complex including the vertex.

void *c3t3.set_index(Vertex_handle v, Index index)*

Sets the index of a vertex.

Queries

Queries on the faces of the embedded complex.

size_type *c3t3.number_of_cells()*

Returns the number of cells that belong to the embedded complex.

size_type *c3t3.number_of_cells(Subdomain_index index)*

Returns the number of cells that belong to the subdomain of the embedded complex with index *index*.

size_type *c3t3.number_of_facets()*

Returns the number of facets that are surface facets, i. e. belong to some surface patch of the embedded complex.

size_type *c3t3.number_of_facets(Surface_patch_index index)*

Returns the number of facets that belong to the surface patch of the embedded complex with index *index*.

bool *c3t3.is_in_complex(Cell_handle c)*

Returns *true* iff the cell *c* belongs to the 3D complex.

bool *c3t3.is_in_complex(Facet f)*

Returns *true* iff the facet *f* belongs to the boundary 2D complex.

bool *c3t3.is_in_complex(Cell_handle c, int i)*

Same as above with *f=(c,i)*.

Queries on the identifier of the face complex including triangulation cells, facets and vertices.

Subdomain_index *c3t3.subdomain_index(Cell_handle c)*

Returns the index of the subdomain containing the cell *c*. The default subdomain index is returned if the cell *c* does not belong to the embedded complex.

Surface_patch_index *c3t3.surface_patch_index(Facet f)*

For a surface facet, returns the index of the surface patch containing the facet. The default *Surface_patch_index* value is returned if the facet is not a surface facet.

Surface_patch_index *c3t3.surface_patch_index(Cell_handle c, int i)*

Same as above with *f=(c,i)*.

int *c3t3.in_dimension(Vertex_handle v) const*

Returns the dimension of the vertex *v*.

Index *c3t3.index(Vertex_handle v) const*

Returns the index of the vertex *v*.

Traversal of the complex

The data structure provides iterators to visit the cells and facets of the complex. All those iterators are bidirectional and non mutable.

Cells_in_complex_iterator

c3t3.cells_in_complex_begin()

Returns a *Cell_in_complex_iterator* to visit the cells of the triangulation contained in the embedded complex.

Cells_in_complex_iterator

c3t3.cells_in_complex_end()

Returns the past-the-end iterator for the above iterator.

Cells_in_complex_iterator

c3t3.cells_in_complex_begin(Subdomain_index index)

Returns a *Cell_in_complex_iterator* to visit the cells of the triangulation which belong to the approximation of subdomain of index *index*.

Cells_in_complex_iterator

c3t3.cells_in_complex_end(Subdomain_index index)

Returns the past-the-end iterator for the above iterator.

Facets_in_complex_iterator

c3t3.facets_in_complex_begin()

Returns a *Facet_in_complex_iterator* to visit the facets in the surface patches of the embedded complexes.

Facets_in_complex_iterator

c3t3.facets_in_complex_end()

Returns the past-the-end iterator for the above iterator.

Facets_in_complex_iterator

c3t3.facets_in_complex_begin(Surface_patch_index index)

Returns a *Facet_in_complex_iterator* to visit the facets of the triangulation which which belong to the approximation of surface patch of index *index*.

Facets_in_complex_iterator

c3t3.facets_in_complex_end(Surface_patch_index index)

Returns the past-the-end iterator for the above iterator.

Has Models

CGAL::Mesh_complex_3_in_triangulation_3<Tr,CornerIndex,CurveSegmentIndex>

See Also

MeshDomain_3 . . . . page 3168
MeshComplexWithFeatures_3InTriangulation_3 . . . . page 3164
CGAL::make_mesh_3 . . . . page 3217

Concept

MeshComplexWithFeatures_3InTriangulation_3

Definition

The concept MeshComplexWithFeatures_3InTriangulation_3 describes a data structure to represent and maintain a 3D complex embedded in a 3D triangulation. The concept MeshComplexWithFeatures_3InTriangulation_3 refines the minimal concept *MeshComplex_3InTriangulation_3*, designed to represent 3D complexes having only faces with dimension 2 and 3. Therefore, the concept MeshComplexWithFeatures_3InTriangulation_3 may represent embedded complexes including *features*, i.e. faces with dimension 0 and 1.

The data structure includes a 3D triangulation which is itself a 3D complex. To distinguish the faces of the embedded 3D complex from the faces of the triangulation, we call respectively *subdomains*, *surface patches* *curve segments* and *corners* the faces of the complex with respective dimensions 3, 2, 1 and 0. The triangulations faces are called respectively cells, facets, edges and vertices.

Each subdomain of the embedded 3D complex is a union of triangulation cells. Likewise, each surface patch is a union of triangulation facets and each curve segment is a union of triangulation edges. The corners form a subset of the triangulation vertices. Note that subdomains, surface patches and and curved segments are not necessarily connected. Likewise each corner may be related to several mesh vertices. Triangulation facets that belong to some surface patch are called surface facets.

The concept MeshComplexWithFeatures_3InTriangulation_3 allows us to mark and retrieve the cells of the triangulation belonging to the subdomains, the facets of the triangulation belonging to surface patches, the edges belonging to curve segments and the vertices that are corners of the embedded complex.

Within the mesh generation functions, the concept MeshComplexWithFeatures_3InTriangulation_3 is the concept describing the data structure used to maintain the current approximation of the input domain. At the end of the meshing process, the data structure encodes the resulting mesh. In particular, each subdomain (resp. surface patch) of the input domain is then approximated by a subdomain (resp. a surface patch) of the embedded complex while the curve segments and corners represent the 1 and 0-dimensional features of the input complex.

Refines

MeshComplex_3InTriangulation_3

Types

MeshComplexWithFeatures_3InTriangulation_3:: Curve_segment_index

A type for indexes of curve segment. The type must match the type *MeshDomainWithFeatures_3::Curve_segment_index* when the concept is used for mesh generation.

MeshComplexWithFeatures_3InTriangulation_3:: Corner_index

A type for indexes of corners. The type must match the type *MeshDomainWithFeatures_3::Corner_index* when the concept is used for mesh generation.

MeshComplexWithFeatures_3InTriangulation_3:: Edges_in_complex_iterator

An iterator type to visit the edges of the triangulation belonging to curve segments.

MeshComplexWithFeatures_3InTriangulation_3:: Vertices_in_complex_iterator

An iterator type to visit the vertices of the triangulation that are corners of the embedded complex.

Modifiers

void *c3t3.add_to_complex(Edge e, Curve_segment_index index)*

Adds edge *e* as an element of the curve segment with index *index*.

void *c3t3.add_to_complex(Vertex_handle v1, Vertex_handle v2, Curve_segment_index index)*

Same as above with *e=(v1,v2)*.

void *c3t3.add_to_complex(Vertex_handle v, Corner_index index)*

Marks vertex *v* as a corner with index *index*.

void *c3t3.remove_from_complex(Edge e)*

Removes edge *e* from the embedded complex.

void *c3t3.remove_from_complex(Vertex_handle v1, Vertex_handle v2)*

Same as above with *e=(v1,v2)*.

void *c3t3.remove_from_complex(Vertex_handle v)*

Removes vertex *v* from the embedded complex.

Queries

Queries on the 1D complex and 0D complex.

size_type *c3t3.number_of_edges() const*

Returns the number of edges which belong to curve segments.

size_type *c3t3.number_of_edges(Curve_segment_index index) const*

Returns the number of edges which belong to curve segment with index *index*.

size_type *c3t3.number_of_corners() const*

Returns the number of vertices which are corners of the complex.

size_type *c3t3.number_of_corners(Corner_index index) const*

Returns the number of vertices which are corners of the complex with index *index*.

bool *c3t3.is_in_complex(Edge e) const*

Returns *true* iff edge *e* belongs to some curve segment.

bool *c3t3.is_in_complex(Vertex_handle v1, Vertex_handle v2) const*

Same as above with *e=(v1,v2)*.

bool *c3t3.is_in_complex(Vertex_handle v) const*

Returns *true* if vertex *v* is a corner.

Curve_segment_index

c3t3.curve_segment_index(Edge e)

Returns *Curve_segment_index* of edge *e*. The default *Curve_segment_index* value is returned if edge *e* does not belong to any curve segment.

Curve_segment_index

c3t3.curve_segment_index(Vertex_handle v1, Vertex_handle v2)

Same as above with *e=(v1,v2)*.

Corner_index *c3t3.corner_index(Vertex_handle v)*

Returns *Corner_index* of vertex *v*. The default *Corner_index* value is returned if vertex *v* is not a corner of the complex.

Traversal of the complex

Edges_in_complex_iterator

c3t3.edges_in_complex_begin() const

Returns an *Edges_in_complex_iterator* to visit the edges of the triangulation belonging to curve segments.

Edge_in_complex_iterator

c3t3.edges_in_complex_end() const

Returns the past-the-end iterator for the above iterator.

Edges_in_complex_iterator

c3t3.edges_in_complex_begin(Curve_segment_index index) const

Returns an *Edges_in_complex_iterator* to visit the edges of the triangulation belonging to curve segments of index *index*.

Edge_in_complex_iterator

c3t3.edges_in_complex_end(Curve_segment_index index) const

Returns the past-the-end iterator for the above iterator.

template <typename OutputIterator>
OutputIterator *c3t3.adjacent_vertices_in_complex(Vertex_handle v, OutputIterator out) const*

Fills *out* with the vertices of the triangulation that are adjacent to vertex *v* through an edge belonging to some curve segment. The value type of *out* must be *std::pair<Vertex_handle,Curve_segment_index>*.
Precondition: *c3t3.in_dimension(v) < 2*

Vertices_in_complex_iterator

c3t3.vertices_in_complex_begin() const

Returns a *Vertices_in_complex_iterator* to visit the vertices of the triangulation that are corners.

Vertices_in_complex_iterator

c3t3.vertices_in_complex_end() const

Returns the past-the-end iterator for the above iterator.

Vertices_in_complex_iterator

c3t3.vertices_in_complex_begin(Corner_index index) const

Returns a *Vertices_in_complex_iterator* to visit the vertices of the triangulation that are corners of index *index*.

Vertices_in_complex_iterator

c3t3.vertices_in_complex_end(Corner_index index) const

Returns the past-the-end iterator for the above iterator.

Has Models

Mesh_complex_3_in_triangulation_3<Tr,CornerIndex,CurveSegmentIndex>

See Also

MeshComplex_3InTriangulation_3 . . . . page 3158
MeshDomainWithFeatures_3 . . . . page 3171

Concept

MeshDomain_3

Definition

The concept MeshDomain_3 describes the knowledge required on the object to be discretized. The concept MeshDomain_3 is the concept to be used when the input domain does not have 0 or 1-dimensional features that need to be accurately approximated by the mesh. In such a case, the queries issued by the meshing process concern only the faces of the input domain with dimension 3 and 2, that are respectively called *subdomains* and *surface patches*.

More specifically the concept MeshDomain_3 provides a method to localize a point with respect to the input domain and its subdomains. Moreover, as the concept *SurfaceMeshTraits_3*, it also provides predicates to test whether a query segment (or a ray, or a line) intersects the boundary of the domain or of the subdomains, and constructors to compute some intersection point if any. It also includes a method able to provide a small set of initial points on the boundary.

In the following we consider only proper intersection with the domain and subdomain boundaries. A segment, ray or line is said to intersect properly the domain boundary (resp. a subdomain boundary) if it includes points which are strictly inside and strictly outside the domain (resp. the subdomain).

Types

MeshDomain_3:: R	Geometric traits class. This type is defined to ensure compatibility with *CGAL::Kernel_traits<T>*.
MeshDomain_3:: Point_3	Point type.
MeshDomain_3:: Segment_3	Segment type.
MeshDomain_3:: Ray_3	Ray type.
MeshDomain_3:: Line_3	Line type.
typedef CGAL::Tag_false *Has_features;*	A type to distinguish *MeshDomain_3* models from *MeshDomainWithFeatures_3* models.
MeshDomain_3:: Subdomain_index	Type of indices for subdomains of the input domain. Must be a model of CopyConstructible, Assignable, DefaultConstructible and EqualityComparable. The default constructed value must match the label of the exterior of the domain (which contains at least the unbounded component).
MeshDomain_3:: Surface_patch_index	Type of indices for surface patches (boundaries and interfaces) of the input domain. Must be a model of CopyConstructible, Assignable, DefaultConstructible and EqualityComparable. The default constructed value must be the index value assigned to a non surface facet.
MeshDomain_3:: Index	Type of indices to be stored at mesh vertices to characterize the lowest dimensional face of the input complex on which the vertex lies. Must be a model of CopyConstructible, Assignable, DefaultConstructible and EqualityComparable.

typedef CGAL::cpp0x::tuple<Point_3, Index, int>

Intersection;	Return type of *Construct_intersection* queries. *int* represents the dimension of the lower dimensional face of the input complex on which the intersection point lies and *Index* is the index of this face.

MeshDomain_3:: Construct_initial_points	A function object to construct a set of initial points on the surface of the domain. Provides the following operators: *template<typename OutputIterator>* *OutputIterator operator()(OutputIterator pts)* *template<typename OutputIterator>* *OutputIterator operator()(int n, OutputIterator pts)* Those two operators output a set of (n) surface points to the output iterator *pts*, as objects of type *std::pair<Point_3, Index>*. If n is not given, the functor must provide enough points to initialize the mesh generation process.
MeshDomain_3:: Is_in_domain	A function object to query whether a point is in the input domain or not. In the positive case, it outputs the subdomain which includes the query point. Provides the operator: *boost::optional<Subdomain_index> operator()(Point_3 p)*
MeshDomain_3:: Do_intersect_surface	A function object which answers intersection queries between the surface patches of the domain and objects of type *Segment_3*, *Ray_3* or *Line_3*. Provides the operators: *boost::optional<Surface_patch_index> operator()(Segment_3 s)* *boost::optional<Surface_patch_index> operator()(Ray_3 r)* *boost::optional<Surface_patch_index> operator()(Line_3 l)* The return type of the operators tell whether or not the query intersects a surface patch. In the positive case, it provides (through operator*()) the *Surface_patch_index* of one of the intersected surface patches.
MeshDomain_3:: Construct_intersection	A function object to construct the intersection between an object of type *Segment_3*, *Ray_3* or *Line_3* and an interface. Provides the operators: *Intersection operator()(Segment_3 s)* *Intersection operator()(Ray_3 r)* *Intersection operator()(Line_3 l)* *Precondition*: do_intersect_surface(s/r/l) == true

Operations

The following functions give access to the function objects:

Construct_initial_points	
	domain.construct_initial_points_object()
Is_in_domain	*domain.is_in_domain_object()*
Do_intersect_surface	*domain.do_intersect_surface_object()*

Construct_intersection

domain.construct_intersection_object()

These methods are designed to convert indices:

Index	*domain.index_from_surface_patch_index(Surface_patch_index surface_patch_index)*	
		Returns the index to be stored at a vertex lying on the surface patch identified by *surface_patch_index*.
Index	*domain.index_from_subdomain_index(Subdomain_index subdomain_index)*	
		Returns the index to be stored at a vertex lying in the subdomain identified by *subdomain_index*.
Surface_patch_index	*domain.surface_patch_index(Index index)*	
		Returns the *Surface_patch_index* of the surface patch where lies a vertex with dimension 2 and index *index*.
Subdomain_index	*domain.subdomain_index(Index index)*	
		Returns the index of the subdomain containing a vertex with dimension 3 and index *index*.

Has Models

Polyhedral_mesh_domain_3<Polyhedron,IGT,TriangleAccessor>
Implicit_mesh_domain_3<Function,BGT>
Labeled_image_mesh_domain_3<Image,BGT>

See Also

MeshVertexBase_3 .... page 3189
MeshCellBase_3 .... page 3187
CGAL::make_mesh_3 .... page 3217
CGAL::refine_mesh_3 .... page 3220

MeshDomainWithFeatures_3

Concept

Definition

The concept MeshDomainWithFeatures_3 refines the concept *MeshDomain_3*. While the concept *MeshDomain_3* only exposes the 2-dimensional and 3-dimensional features of the domain through different queries, the concept MeshDomainWithFeatures_3 also exposes 0 and 1-dimensional features. The exposed features of the domain are respectively called subdomains, surface patches, curve segments and corners according to their respective dimensions 3,2,1 and 0.

Refines

MeshDomain_3

Types

typedef CGAL::Tag_true

Has_features;	A type to distinguish *MeshDomain_3* models from *MeshDomainWithFeatures_3* models.
MeshDomainWithFeatures_3:: FT	Numerical type.
MeshDomainWithFeatures_3:: Curve_segment_index	
	Type of indices for curve segments (1-dimensional features) of the input domain. Must be a model of CopyConstructible, Assignable, DefaultConstructible and LessThanComparable. The default constructed value must be the value of an edge which does not approximate a 1-dimensional feature of the input domain.
MeshDomainWithFeatures_3:: Corner_index	Type of indices for corners (i.e. 0-dimensional features) of the input domain. Must be a model of CopyConstructible, Assignable, DefaultConstructible and LessThanComparable.

Operations

Each connected component of a curve segment is assumed to be oriented. The orientation is defined by the ordering of the two incident corners at the origin and endpoint. Therefore it is possible to defined the signed geodesic distance between two ordered points on the same connected component of a curve segment. A cycle is a connected component of a curve segment incident to 0 or 1 corner.

Point_3 *md.construct_point_on_curve_segment(Point_3 p,*
Curve_segment_index ci,

FT d) const

Returns a point on the curve segment with index *ci* at signed geodesic distance *d* from point *p*.
Precondition: Point *p* is supposed to be on curve segment *ci*. If *d > 0*, the signed geodesic distance from *p* to the endpoint of the connected component of *ci* including *p*, should be greater than *d*. If *d < 0*, the signed geodesic distance from *p* to the origin of the connected component should be less than *d* from the origin of the connected component.

Note that *construct_point_on_curve segment* is assumed to return a uniquely defined point. Therefore it is not possible to handle as a single curve segment, a singular curve with several branches incident to the same point.

Queries

FT *md.geodesic_distance(Point_3 p, Point_3 q, Curve_segment_index ci) const*

Returns the signed geodesic distance between points *p* and *q* along the input curve segment with index *ci*.
Precondition: Points *p* and *q* belong to the same connected component of the curve segment with index *ci*.

CGAL::Sign *md.distance_sign_along_cycle(Point_3 p, Point_3 q, Point_3 r, Curve_segment_index ci) const*

Returns *CGAL::POSITIVE* if the signed geodesic distance from *p* to *q* on the way through *r* along cycle with index *ci* is positive, *CGAL::NEGATIVE* if the distance is negative, and *CGAL::ZERO* if *(p = q = r)*.
Precondition: Points *p*, *q* and *r* belongs to the same connected component of curve segment *ci* and this component is a cycle.

bool *md.is_cycle(Point_3 p, Curve_segment_index ci) const*

Returns *true* if the connected component of curve segment *ci* including point *p* is a cycle.

Retrieval of the input features and their incidences

template <typename OutputIterator>
OutputIterator *md.get_corners(OutputIterator corners) const*

Fills *corners* with the corners of the input domain. *corners* value type must be *std::pair<Corner_index,Point_3>*.

template <typename OutputIterator>

OutputIterator *md.get_curve_segments(OutputIterator curves) const*

Fills *curves* with the curve segments of the input domain. *curves* value type must be *CGAL::cpp0x::tuple<Curve_segment_index,std::pair<Point_3,Index>,std::pair<Point_3,Index> >*. If the curve segment corresponding to an entry in curves is not a cycle, the pair of associated points should belong to two corners incident on the curve segment. If it is a cycle, then the same *Point_3* should be given twice and must be any point on the cycle. The *Index* values associated to the points are their indices w.r.t. their dimension.

bool *md.are_incident_surface_patch_curve_segment(Surface_patch_index spi, Curve_segment_index csi)*

Returns *true* if the curve segment with index *csi* is incident to the surface patch with index *spi*.

bool *md.are_incident_surface_patch_corner(Surface_patch_index spi, Corner_index ci)*

Returns *true* if the corner with index *ci* is incident to the surface patch with index *spi*.

Indices converters

Index *md.index_from_curve_segment_index(Curve_segment_index curve_segment_index) const*

Returns the index to be stored at a vertex lying on the curve segment identified by *curve_segment_index*.

Curve_segment_index *md.curve_segment_index(Index index) const*

Returns the *Curve_segment_index* of the curve segment where lies a vertex with dimension 1 and index *index*.

Index *md.index_from_corner_index(Corner_index corner_index) const*

Returns the index to be stored at a vertex lying on the corner identified by *corner_index*.

Corner_index *md.corner_index(Index index) const*

Returns the *Corner_index* of the corner where lies a vertex with dimension 0 and index *index*.

Has Models

Mesh_domain_with_polyline_features_3<MeshDomain_3>
Polyhedral_mesh_domain_with_features_3<IGT>

See Also

MeshDomain_3 . . . . page 3168

MeshCriteria_3

Definition

The Delaunay refinement process involved in the template functions *make_mesh_3* and *refine_mesh_3* is guided by a set of elementary refinement criteria that concern either mesh tetrahedra or surface facets. The refinement criteria for tetrahedra are described through the concept *MeshCellCriteria_3* while the refinement criteria for surface facets are described by the concept *MeshFacetCriteria_3*. The concept MeshCriteria_3 encapsulates these concepts.

Types

MeshCriteria_3:: Facet_criteria	Functor that describes the criteria for surface facets. This type must be a model of the concept *MeshFacetCriteria_3*.
MeshCriteria_3:: Cell_criteria	Functor that describes the criteria for mesh tetrahedra. This type must be a model of the concept *MeshCellCriteria_3*.

Access Functions

Facet_criteria *criteria.facet_criteria_object()*

Returns the facet criteria.

Cell_criteria *criteria.cell_criteria_object()*

Returns the cell criteria.

Has Models

Mesh_criteria_3<Tr>

See Also

MeshFacetCriteria_3 . . . . page 3178
MeshCellCriteria_3 . . . . page 3177
CGAL::make_mesh_3 . . . . page 3217
CGAL::refine_mesh_3 . . . . page 3220
MeshCriteriaWithFeatures_3 . . . . page 3176

Concept

MeshCriteriaWithFeatures_3

Definition

The concept MeshCriteriaWithFeatures_3 refines the concept *MeshCriteria_3*. The concept *MeshCriteria_3* encapsulates the concepts *MeshCellCriteria_3* and *MeshFacetCriteria_3* describing the refinement criteria for, respectively, mesh cells and surface facets. For domains with features, the concept MeshCriteriaWithFeatures_3 additionnally encapsulates the concept *MeshEdgeCriteria_3*, that describes the requirements, in terms of sizing, for the discretization of the domain 1-dimensional features.

Refines

MeshDomain_3

Types

MeshCriteriaWithFeatures_3:: Edge_criteria Functor that describes the criteria for the mesh edges that discretize the input domain 1-dimensional features. This type must be a model of the concept *MeshEdgeCriteria_3*.

Access Functions

Edge_criteria *criteria.edge_criteria_object()*

Returns the edge criteria.

Has Models

Mesh_criteria_3<Tr>

See Also

MeshCriteria_3 . . . . page 3175
MeshEdgeCriteria_3 . . . . page 3180
MeshFacetCriteria_3 . . . . page 3178
MeshCellCriteria_3 . . . . page 3177
CGAL::make_mesh_3 . . . . page 3217
CGAL::refine_mesh_3 . . . . page 3220

MeshCellCriteria_3

Definition

The Delaunay refinement process involved in the template functions *make_mesh_3* and *refine_mesh_3* is guided by a set of elementary refinement criteria that concern either mesh tetrahedra or surface facets. The concept MeshCellCriteria_3 describes the types that handle the refinement criteria for mesh tetrahedra.

Types

MeshCellCriteria_3:: Cell_handle	Handle type for the cells of the triangulation. Must match the *Cell_handle* type in the triangulation type used by the mesh generation function.
MeshCellCriteria_3:: Cell_quality	Type representing the quality of a cell. Must be a model of CopyConstructible and LessThanComparable. Between two cells, the one which has the lower quality must have the lower *Cell_quality*.
MeshCellCriteria_3:: Is_cell_bad	Type representing if a cell is bad or not. Must be convertible to *bool*. If it converts to *true* then the cell is bad, otherwise the cell is good with regard to the criteria. In addition, an object of this type must contain an object of type *Cell_quality* if it represents a bad cell. *Cell_quality* must be accessible by *operator*()*. Note that *boost::optional< Cell_quality>* is a natural model of this concept.

Operations

Is_cell_bad   *cell_criteria(Cell_handle c)*

Returns *Is_cell_bad* value of cell *c*.

Has Models

Mesh_cell_criteria_3<Tr>

See Also

MeshEdgeCriteria_3 . . . . page 3180
MeshFacetCriteria_3 . . . . page 3178
MeshCriteria_3 . . . . page 3175
CGAL::make_mesh_3 . . . . page 3217
CGAL::refine_mesh_3 . . . . page 3220

Functor
Concept

MeshFacetCriteria_3

Definition

The Delaunay refinement process involved in the template functions *make_mesh_3* and *refine_mesh_3* is guided by a set of elementary refinement criteria that concern either mesh tetrahedra or surface facets. The concept MeshFacetCriteria_3 describes the types that handle the refinement criteria for surface facets.

Types

MeshFacetCriteria_3:: Facet

Type for the facets of the triangulation. Must match the *Facet* type in the triangulation type used by the mesh generation function.

MeshFacetCriteria_3:: Cell_handle

Handle type for the cells of the triangulation. Must match the *Cell_handle* type in the triangulation type used by the mesh generation function.

MeshFacetCriteria_3:: Facet_quality

Type representing the quality of a facet. Must be a model of CopyConstructible and LessThanComparable. Between two facets, the one which has the lower quality must have the lower *Facet_quality*.

MeshFacetCriteria_3:: Is_facet_bad

Type representing if a facet is bad or not. This type must be convertible to *bool*. If it converts to *true* then the facet is bad, otherwise the facet is good with regard to the criteria.
In addition, an object of this type must contain an object of type *Facet_quality* if it represents a bad facet. *Facet_quality* must be accessible by *operator*()*. Note that *boost::optional<Facet_quality>* is a natural model of this concept

Operations

Is_facet_bad *facet_criteria(Facet f)*

Returns *Is_facet_bad* value of facet f.

Is_facet_bad *facet_criteria(Cell_handle c, int i)*

Same as above with *f=(c,i)*.

Has Models

Mesh_facet_criteria_3<Tr>

See Also

MeshCellCriteria_3 . . . . page 3177
MeshEdgeCriteria_3 . . . . page 3180
MeshCriteria_3 . . . . page 3175
CGAL::make_mesh_3 . . . . page 3217
CGAL::refine_mesh_3 . . . . page 3220

MeshEdgeCriteria_3

Definition

The function object concept MeshEdgeCriteria_3 is designed to drive the process which samples the 1-dimensional features of the domain. It provides an upper bound for the distance between two protecting ball centers that are consecutive on a 1-feature.

Types

MeshEdgeCriteria_3:: Point_3	Point type. Must match the *Point_3* type in the triangulation type used by the mesh generation function.
MeshEdgeCriteria_3:: Edge	Type for edges of the triangulation. Must match the *Edge* type in the triangulation type used by the mesh generation function.
MeshEdgeCriteria_3:: FT	Numerical type.

Operations

bool *edge_criteria(Edge e)*

Returns *true* if edge *e* does not fulfill the criteria.

FT *edge_criteria.sizing_field(Point_3 p)*

Returns the value of the sizing field (i.e. the maximum edge length) at point *p*.

Has Models

Mesh_edge_criteria_3<Tr>

See Also

MeshCellCriteria_3 . . . . page 3177
MeshFacetCriteria_3 . . . . page 3178
MeshCriteria_3 . . . . page 3175
MeshCriteriaWithFeatures_3 . . . . page 3176

BisectionGeometricTraits_3

Concept

Definition

The concept BisectionGeometricTraits_3 describes a geometric traits class that provides the basic types and operations to implement a model of *MeshDomain_3* based solely on intersection detections. Points in the non-empty intersections are herein computed by bisection.

Such traits class is relevant when intersection detections can be performed efficiently. For instance, when bounding surfaces are implicitly described by a function (such as an isosurface of a 3D function from $\mathbb{R}^3$ to $\mathbb{R}$), the do-intersect predicate with a segment is computed by evaluations of the function values at both end points of the segment.

Types

BisectionGeometricTraits_3:: FT	Numerical type. Must be a model of *SqrtFieldNumberType* and constructible from a *double*.
BisectionGeometricTraits_3:: Point_3	The point type. Must have a constructor *Point_3(FT, FT, FT)*.
BisectionGeometricTraits_3:: Segment_3	Segment type.
BisectionGeometricTraits_3:: Ray_3	Ray type.
BisectionGeometricTraits_3:: Line_3	Line type.
BisectionGeometricTraits_3:: Vector_3	Vector type.
BisectionGeometricTraits_3:: Sphere_3	Sphere type.

BisectionGeometricTraits_3:: Compute_scalar_product_3

Function object providing the operator
FT operator()(Vector_3 v, Vector_3 w) which returns the scalar (inner) product of the two vectors *v* and *w*.

BisectionGeometricTraits_3:: Compute_squared_distance_3

Function object providing the operator
FT operator()(Point_3, Point_3) which returns the squared distance between two points.

BisectionGeometricTraits_3:: Compute_squared_radius_3

Function object providing the operator
FT operator()(Sphere_3 s) which returns the squared radius of *s*.

BisectionGeometricTraits_3:: Construct_center_3

Function object providing the operator
Point_3 operator()(Sphere_3 s) which returns the center of the sphere *s*.

BisectionGeometricTraits_3:: Construct_midpoint_3

Function object providing the operator
Point_3 operator()(Point_3 p, Point_3 q) which computes the midpoint of the segment *pq*.

BisectionGeometricTraits_3:: Construct_point_on_3

Function object providing the following operators:
Point_3 operator()(Line_3 l,int i); which returns an arbitrary point on *l*. It holds *point(i) == point(j)*, iff *i==j*. Furthermore, is directed from *point(i)* to *point(j)*, for all $i < j$.
Point_3 operator()(Ray_3 r,int i); which returns a point on *r*. *point(0)* is the source, *point(i)*, with $i > 0$, is different from the source.
Precondition: $i \geq 0$.
Point_3 operator()(Segment_3 s,int i); which returns either source or target of *s*: *point(0)* returns the source of *s*, *point(1)* returns the target of *s*. Parameter *i* is taken modulo 2, which gives easy access to the other end point.

BisectionGeometricTraits_3:: Construct_segment_3

Function object providing the operator
Segment_3 operator()(Point_3 p, Point_3 q); which returns a segment with source *p* and target *q*, directed from the source to the target.

BisectionGeometricTraits_3:: Construct_scaled_vector_3

Function object providing the operator
Vector_3 operator()(Vector_3 v, FT scale) which returns the vector *v* scaled by a factor *scale*.

BisectionGeometricTraits_3:: Construct_translated_point_3

Function object providing the operator
Point_3 operator()(Point_3 p, Vector_3 v) which returns the point obtained by translating *p* by the vector *v*.

BisectionGeometricTraits_3:: Construct_vector_3

Function object providing the operator
Vector_3 operator()(Point_3 a, Point_3 b) which returns the vector *b-a*.

BisectionGeometricTraits_3:: Has_on_bounded_side_3

Function object providing the operator
bool operator()(Sphere_3 s, Point_3 p); which returns true iff *p* lies on the bounded side of *s*.

Operations

The following functions give access to the predicate and construction objects:

Compute_scalar_product_3	*traits.compute_scalar_product_3_object()*
Compute_squared_distance_3	*traits.compute_squared_distance_3_object()*
Compute_squared_radius_3	*traits.compute_squared_radius_3_object()*
Construct_center_3	*traits.construct_center_3_object()*
Construct_midpoint_3	*traits.construct_midpoint_3_object()*
Construct_point_on_3	*traits.construct_point_on_3_object()*
Construct_scaled_vector_3	*traits.construct_scaled_vector_3_object()*
Construct_segment_3	*traits.construct_segment_3_object()*
Construct_translated_point_3	*traits.construct_translated_point_3_object()*
Construct_vector_3	*traits.construct_vector_3_object()*
Has_on_bounded_side_3	*traits.has_on_bounded_side_3_object()*

Has Models

Any CGAL Kernel.

See Also

ImplicitSurfaceTraits_3 . . . . page 3057
IntersectionGeometricTraits_3 . . . . page 3184
CGAL::Implicit_mesh_domain_3<Function,BGT> . . . . page 3207
CGAL::Labeled_image_mesh_domain_3<Image,BGT> . . . . page 3212

Concept

IntersectionGeometricTraits_3

Definition

The concept IntersectionGeometricTraits_3 provides types and functors required to implement a model of *MeshDomain_3*, when the domain is described by a simplicial surface mesh forming its boundary. The concept IntersectionGeometricTraits_3 mainly provides the detection and construction of intersections between segments and triangles.

Types

IntersectionGeometricTraits_3:: Point_3	Point type.
IntersectionGeometricTraits_3:: Segment_3	Segment type.
IntersectionGeometricTraits_3:: Triangle_3	Triangle type.

IntersectionGeometricTraits_3:: Do_intersect_3

Function object that detects an intersection between a 3D segment and a 3D triangle. Provides the operators:
bool operator()(Segment_3 seg, Triangle_3 tr)
bool operator()(Triangle_3 tr, Segment_3 seg)
which return *true*, iff the triangle and the segment have a non empty intersection.

IntersectionGeometricTraits_3:: Intersect_3

Function object that constructs the intersection between a 3D segment and a 3D triangle. Provides the operators:
CGAL::Object operator()(Segment_3 seg, Triangle_3 tr)
CGAL::Object operator()(Triangle_3 tr, Segment_3 seg)
which computes as a *CGAL::Object* the intersection between the triangle and the segment. *CGAL::Object* is either a point, a segment or an empty object.

Operations

Do_intersect_3 *traits.do_intersect_3_object()*

Returns the intersection detection functor.

Intersect_3 *traits.intersect_3_object()*

Returns the intersection constructor.

Has Models

Any instantiation of *CGAL::Kernel* is a model of this traits concept.

See Also

BisectionGeometricTraits_3 . . . . page 3181
CGAL::Polyhedral_mesh_domain_3<Polyhedron,IGT,TriangleAccessor> . . . . page 3209

Concept

LabeledImage_3

Definition

The concept LabeledImage_3 describes the requirements for the second template parameter of the class *Labeled_image_mesh_domain_3<Image,BGT>* which represents mesh domains defined by 3D labeled images. A 3D labeled image is a 3D array of elements of an integral type *Type*. *Type* can be *bool*, *char*, *short*, *int*, or *long* (signed or not). Such an array is associated to a 3D axis-aligned regular grid, in $\mathbb{R}^3$. A cell of this grid is denoted by *voxel*. A voxel is an iso-cuboid of size *vx()*, *vy()*, and *vz()*.

Types

LabeledImage_3:: Type	Type of voxel data. Must be an integral type.
LabeledImage_3:: RT	Ring number type.

Operations

int	*image.xdim()*	First dimension of the 3D array, i.e., the number of voxels along the x coordinate axis.
int	*image.ydim()*	Second dimension of the 3D array.
int	*image.zdim()*	Third dimension of the 3D array.
RT	*image.vx()*	Size of each voxel along x coordinate axis.
RT	*image.vy()*	Size of each voxel along y coordinate axis.
RT	*image.vz()*	Size of each voxel along z coordinate axis.
*const Type**	*image.data()*	Pointer to the first element of the 3D image. The size of the array must be *xdim()*×*ydim()*×*zdim()*.

Has Models

CGAL::Image_3<Kernel, T>, for any CGAL kernel *K* and any integral type *T*.

MeshCellBase_3

Definition

The concept MeshCellBase_3 describes the requirements for the *Cell* type of the triangulation used in the 3D mesh generation process. The type MeshCellBase_3 refines the concept *RegularTriangulationCellBase_3*. The concept MeshCellBase_3 includes a way to store and retrieve if a given cell of the triangulation is inside the domain or not and which subdomain it belongs to in case of a multi-domain.

Moreover, this concept adds four markers per cell to mark the facets of the triangulation that are surface facets, and four additional helper markers used in some operations to mark for instance the facets that have been visited.

This concept also provides storage for the centers of Delaunay surface balls. Each surface facet has a Delaunay surface ball, i. e. a circumscribing ball, centered on an input complex surface patch, and empty of triangulation vertices. In the following we call *surface center* of a surface facet, the center of its biggest Delaunay surface ball.

Refines

RegularTriangulationCellBase_3

Types

MeshCellBase_3:: Point — Point type, required to match the point type of the 3D triangulation in which the mesh is embedded.

MeshCellBase_3:: Subdomain_index; — Type of indices for cells of the input complex. Must match the type *MeshDomain_3::Subdomain_index*.

MeshCellBase_3:: Surface_patch_index; — Type of indices for surface patches of the input complex. Must match the type *MeshDomain_3::Surface_patch_index*.

Operations

Subdomain_index *cell.subdomain_index()*

Returns the index of the input subdomain that contains the cell *cell* of the triangulation.

void *cell.set_subdomain_index(Subdomain_index index)*

Sets the subdomain index of the cell.

bool *cell.is_facet_on_surface(int i)*

returns *true* iff *facet(i)* is a surface facet.

Surface_patch_index	*cell.surface_patch_index(int i)*
	returns *Surface_patch_index* of facet *i*.
void	*cell.set_surface_patch_index(int i, Surface_patch_index index)*
	sets *Surface_patch_index* of facet *i* to *index*.

bool	*cell.is_facet_visited(int i)*
	Returns *true* iff *facet(i)* has been visited.
void	*cell.set_facet_visited(int i, bool b)*
	Marks *facet(i)* as visited if *b* is *true* and non-visited otherwise.

Point	*cell.facet_surface_center(int i)*
	Returns a const reference to the surface center of *facet(i)*.
void	*cell.set_facet_surface_center(int i, Point p)*
	Sets point *p* as the surface center of *facet(i)*.

Has Models

Mesh_cell_base_3<MD,Gt,Cb>

See Also

CGAL::make_mesh_3 . . . . page 3217
MeshDomain_3 . . . . page 3168
Complex_3InTriangulation_3 . . . . page **??**

MeshVertexBase_3

Concept

Definition

The concept MeshVertexBase_3 describes the requirements for the *Vertex* type of the triangulation used by a 3D mesh generation process. The type MeshVertexBase_3 refines both the concept *RegularTriangulationVertexBase_3* and the concept *SurfaceMeshVertexBase_3*. It provides additional members to store and retrieve information about the location of the vertex with respect to the input domain describing the domain to be discretized. More specifically, the concept MeshVertexBase_3 provides read-write access to an integer representing the dimension of the lowest dimensional face of the input 3D complex on which the vertex lies, and to an index characteristic of this face.

Refines

RegularTriangulationVertexBase_3
SurfaceMeshVertexBase_3

Types

MeshVertexBase_3:: Index;	Index type. Must match the type *MeshDomain_3::Index*.
MeshVertexBase_3:: FT;	Numerical type.

Operations

int	*vertex.in_dimension() const*	
		Returns the dimension of the lowest dimensional face of the input 3D complex that contains the vertex.
void	*vertex.set_dimension(int)*	
		Sets the dimension of the lowest dimensional face of the input 3D complex that contains the vertex.
Index	*vertex.index()*	Returns the index of the lowest dimensional face of the input 3D complex that contains the vertex.
void	*vertex.set_index(Index)*	
		Sets the index of the lowest dimensional face of the input 3D complex that contains the vertex.

Internal

FT	*vertex.meshing_info() const*
void	*vertex.set_meshing_info(FT)*

These functions are used internally by mesh optimizers. The user is not encouraged to use them directly as they may change in the future.

Has Models

Mesh_vertex_base_3<MD,Gt,Vb>

See Also

CGAL::make_mesh_3 . . . . page 3217
CGAL::refine_mesh_3 . . . . page 3220
MeshDomain_3 . . . . page 3168

MeshDomainField_3

Functor
Concept

Definition

The concept MeshDomainField_3 describes a scalar field which could be queried at any point of the space.

Types

MeshDomainField_3:: FT	Numerical type.
MeshDomainField_3:: Point_3	Point type.
MeshDomainField_3:: Index	Index type for points. Must match the type *MeshDomain_3::Index*.

Operations

The *operator()* returns the field value at a query point. The field value may depend on the query point location and/or on the input feature including the query point.

FT *msf(Point_3 p, int dimension, Index index)*

returns the value of the sizing field at the point *p*, assumed to be included in the input complex feature with dimension *dimension* and mesh vertex index *index*.

Has Models

Mesh_constant_field_3<Gt,Index>

See Also

MeshDomain_3 . . . . page 3168
MeshDomainWithFeatures_3 . . . . page 3171
CGAL::Mesh_edge_criteria_3<Tr> . . . . page 3205
CGAL::Mesh_facet_criteria_3<Tr> . . . . page 3203
CGAL::Mesh_cell_criteria_3<Tr> . . . . page 3202

Concept

MeshPolyline_3

Definition

The concept MeshPolyline_3 implements a container of points designed to represent a polyline (i.e. a sequence of points). Types and functions provided in this concept are such as standard template library containers are natural models of this concept.

Types

MeshPolyline_3:: value_type	Point type. Must match the type *MeshDomain_3::Point_3*.
MeshPolyline_3:: const_iterator	A constant iterator on points. Must be a model of Bidirectional iterator and have *value_type* as value type.

Operations

const_iterator	*polyline.begin()*	Returns an iterator on the first point of the polyline.
const_iterator	*polyline.end()*	Returns the past-the-end iterator for the above iterator.

Has Models

std::vector<Kernel::Point_3> for any Kernel of CGAL is a natural model of this concept.

See Also

CGAL::Mesh_domain_with_polyline_features_3<MeshDomain> .... page **??**

TriangleAccessor_3

Definition

The concept TriangleAccessor_3 represents an accessor to a triangulated polyhedral surface, intersection free and without boundaries.

Types

TriangleAccessor_3:: Triangle_3	Triangle type. Must be a model of DefaultConstructible and CopyConstructible.
TriangleAccessor_3:: Triangle_iterator	Triangle iterator type. Must be a model of InputIterator.
TriangleAccessor_3:: Triangle_handle	Handle to a *Triangle_3*. Must be constructible from *Triangle_iterator*. It may be *Triangle_Iterator* itself.
TriangleAccessor_3:: Polyhedron	Polyhedron type.

Operations

Triangle_iterator *ta.triangles_begin(Polyhedron p)*

Returns a *Triangle_iterator* to visit the triangles of polyhedron *p*.

Triangle_iterator *ta.triangles_end(Polyhedron p)*

Returns the past-the-end iterator for the above iterator.

Triangle_3 *ta.triangle(Triangle_handle h)*

Returns a *Triangle_3* object from handle *h*.

Has Models

Triangle_accessor_3<Polyhedron_3<K>,K>

See Also

CGAL::Polyhedral_mesh_domain_3<Polyhedron,IGT,TriangleAccessor> .... page 3209
CGAL::make_mesh_3 .... page 3217

Class

CGAL::Mesh_complex_3_in_triangulation_3<Tr,CornerIndex,CurveSegmentIndex>

Definition

The class *Mesh_complex_3_in_triangulation_3<Tr,CornerIndex,CurveSegmentIndex>* implements a data structure to store the 3D restricted Delaunay triangulation used by a mesh generation process.

This class is a model of the concept *MeshComplexWithFeatures_3InTriangulation_3*.

Parameters

The template parameter *Tr* can be instantiated with any 3D regular triangulation of CGAL provided that its vertex and cell base class are models of the concepts *MeshVertexBase_3* and *MeshCellBase_3*, respectively.

The template parameter *CornerIndex* is the type of the indices for corners and the template parameter *CurveSegmentIndex* is the type of the indices for curves segments. They must match the *Corner_index* and *Curve_segment_index* types of the model of the *MeshDomainWithFeatures_3* concept used for mesh generation.

Those two last template parameters defaults to *int*, so that they can be ignored if the domain used for mesh generation does not include 0 and 1-dimensionnal features (i.e is a model of the concept *MeshDomain_3*).

#include <CGAL/Mesh_complex_3_in_triangulation_3.h>

Is Model for the Concepts

MeshComplexWithFeatures_3InTriangulation_3

Types

typedef Tr::Vertex::Index

Index; Index type.

typedef Tr::Cell::Surface_patch_index

Surface_patch_index; Surface index type.

typedef Tr::Cell::Subdomain_index

Subdomain_index; Subdomain index type.

typedef CornerIndex *Corner_index;* *Corner_index* type.

typedef CurveSegmentIndex

Curve_segment_index;

Curve_segment_index type.

Operations

void *c3t3.output_to_medit(std::ofstream& os)*

Outputs the mesh to *os* in medit format.

See Also

CGAL::make_mesh_3 . . . page 3217
CGAL::refine_mesh_3 . . . page 3220
MeshComplex_3InTriangulation_3 . . . page 3158
MeshComplexWithFeatures_3InTriangulation_3 . . . page 3164
MeshCellBase_3 . . . page 3187,
MeshVertexBase_3 . . . page 3189

Class

CGAL::Mesh_triangulation_3<MD,Gt>

Definition

The class *Mesh_triangulation_3<MD,Gt>* provides a default triangulation to be used as the 3D triangulation of a mesh generation process.

#include <CGAL/Mesh_triangulation_3.h>

Parameters

Template parameter *MD* stands for a model of *MeshDomain_3*.

Template parameter *Gt* stands for a model of *RegularTriangulationTraits_3* and defaults to *Kernel_traits<MD> ::Kernel*.

Types

Mesh_triangulation_3<MD,Gt>:: type	*CGAL::Regular_triangulation_3* type whose vertex and cell classes are respectively *Mesh_vertex_base_3<MD,Gt>* and *Mesh_cell_base_3<MD,Gt>*.

See Also

CGAL::make_mesh_3 . . . . page 3217
CGAL::Mesh_complex_3_in_triangulation_3<Tr,CornerIndex,CurveSegmentIndex> . . . . page 3194

CGAL::Mesh_vertex_base_3<MD,Gt,Vb>

Class

Definition

The class *Mesh_vertex_base_3<MD,Gt,Vb>* is a model of the concept *MeshVertexBase_3*. It is designed to serve as vertex base class for the 3D triangulation used in a 3D mesh generation process.

Parameters

The template parameter *MD* provides the types of indices used to identify the faces of the input complex. It must be a model of the concept *MeshDomain_3*.

The template parameter *Gt* is the geometric traits class. It must be a model of the concept *RegularTriangulationTraits_3*.

The third parameter *Vb* is the vertex base class. It has to be a model of the concept *TriangulationVertexBase_3* and defaults to *Triangulation_vertex_base_3<Gt>*.

#include <CGAL/Mesh_vertex_base_3.h>

Is Model for the Concepts

MeshVertexBase_3

Inherits From

Vb

See Also

MeshVertexBase_3 . . . . page 3189
CGAL::Mesh_complex_3_in_triangulation_3<Tr,CornerIndex,CurveSegmentIndex> . . . . page 3194

CGAL::Mesh_cell_base_3<MD,Gt,Cb>

Definition

The class *Mesh_cell_base_3<MD,Gt,Cb>* is a model of the concept *MeshCellBase_3*. It is designed to serve as cell base class for the 3D triangulation used in the 3D mesh generation process.

Parameters

The template parameter *MD* provides the types of indices used to identify the faces of the input complex. It has to be a model of the concept *MeshDomain_3*.

The template parameter *Gt* is the geometric traits class. It has to be a model of the concept *RegularTriangulationTraits_3*.

The third parameter *Cb* is the cell base class. It has to be a model of the concept *RegularTriangulationCellBase_3* and defaults to *Regular_triangulation_cell_base_3<Gt>*.

#include <CGAL/Mesh_cell_base_3.h>

Is Model for the Concepts

MeshCellBase_3

Inherits From

Cb

See Also

MeshCellBase_3 .... page 3187
CGAL::Mesh_complex_3_in_triangulation_3<Tr,CornerIndex,CurveSegmentIndex> .... page 3194

CGAL::Mesh_criteria_3<Tr>

Class

Definition

The class *Mesh_criteria_3<Tr>* is a model of both concepts *MeshCriteria_3* and *MeshCriteriaWithFeatures_3*. It gathers the refinement criteria for mesh tetrahedra and surface facets where surface facets are facets in the mesh approximating the domain surface patches. In addition, for domain with exposed 1-dimensional features, the class *Mesh_criteria_3<Tr>* handles the definition of a sizing field to guide the discretization of 1-dimensional features.

#include <CGAL/Mesh_criteria_3.h>

Parameters

The parameter *Tr* has to be instantiated with the type used for *C3T3::Triangulation*, where *C3T3* is the model of *MeshComplex_3InTriangulation_3* used in the mesh generation process, and *C3T3::Triangulation* its nested triangulation type.

Is Model for the Concepts

MeshCriteria_3

Types

typedef Mesh_edge_criteria_3<Tr>

Edge_criteria; The criteria for edges.

typedef Mesh_facet_criteria_3<Tr>

Facet_criteria; The criteria for facets.

typedef Mesh_cell_criteria_3<Tr>

Cell_criteria; The criteria for cells.

Creation

Mesh_criteria_3<Tr> mc(Facet_criteria facet_criteria, Cell_criteria cell_criteria);

Construction from facet and cell criteria. The edge criteria are ignored in this case.

Mesh_criteria_3<Tr> mc(Edge_criteria edge_criteria,
Facet_criteria facet_criteria,
Cell_criteria cell_criteria)

Construction from edge, facet and cell criteria.

template<typename FT, typename ...Fieldi>
Mesh_criteria_3<Tr> mc(Field1 parameters::edge_size = ignored,
FT parameters::facet_angle = ignored,
Field2 parameters::facet_size = ignored,
Field3 parameters::facet_distance = ignored,
Mesh_facet_topology parameters::facet_topology = CGAL::FACET_VERTICES_ON_SURFACE,
FT parameters::cell_radius_edge_ratio = ignored,
Field4 parameters::cell_size = ignored)

Construction from criteria parameters. This constructor uses named parameters (from *Boost.Parameter*) for convenient criteria construction. See a complete description of these parameters below.

The template parameter *FT* should be a model of concept *FieldType*. The template parameters *Fieldi* ($i \in \{1..4\}$) should be either a model of the concept *FieldType* or a model of the concept *MeshDomainField_3*.

The parameters are named parameters and can be passed in any order provided their name is given (see example below). The name of each parameter is the one that is written in the description of the function (e.g. *parameters::facet_size*).

The description of each parameter is as follows:

-*edge_size*: a scalar field (resp. a constant) providing a space varying (resp. a uniform) upper bound for the lengths of curve segment edges.

-*facet_angle*: a lower bound for the angles (in degrees) of the surface mesh facets.
-*facet_size*: a scalar field (resp. a constant) describing a space varying (resp. a uniform) upper-bound or for the radii of the surface Delaunay balls.
-*facet_distance*: a scalar field (resp. a constant) describing a space varying (resp. a uniform) upper bound for the same distance.
-*facet_topology*: the set of topological constraints which have to be verified by each surface facet. The default value is *CGAL::FACET_VERTICES_ON_SURFACE*. See *Mesh_facet_topology* manual page to get all possible values.

-*cell_radius_edge_ratio*: an upper bound for the radius-edge ratio of the mesh tetrahedra.
-*cell_size*: a scalar field (resp. a constant) describing a space varying (resp. a uniform) upper-bound for the circumradii of the mesh tetrahedra.

Note that each size or distance parameter can be specified using two ways: either as scalar field or as a numerical value when the field is uniform.

Each parameter has a special default value *ignored* which means that the corresponding criteria will be ignored. Numerical sizing or distance values, as well as scalar fields should be given in the unit used for coordinates of points in the mesh domain class of the mesh generation process.

Example

```
// Create a Mesh_criteria_3<Tr> object with all cell and facet parameters
    set
Mesh_criteria_3<Tr> criteria (parameters::facet_angle=30,
                              parameters::facet_size=1,
                              parameters::facet_distance=0.1,
```

```
                                parameters::cell_radius_edge_ratio=2,
                                parameters::cell_size=1.5);

// Create a Mesh_criteria_3<Tr> object with size ignored (note that the
    order changed)
Mesh_criteria_3<Tr> criteria (parameters::cell_radius_edge_ratio=2,
                                parameters::facet_angle=30,
                                parameters::facet_distance=0.1);
```

See Also

MeshCriteria_3 . . . page 3175
MeshCriteriaWithFeatures_3 . . . page 3176
MeshCellCriteria_3 . . . page 3177
MeshEdgeCriteria_3 . . . page 3180
MeshFacetCriteria_3 . . . page 3178
MeshDomainField_3 . . . page 3191
CGAL::Mesh_cell_criteria_3<Tr> . . . page 3202
CGAL::Mesh_edge_criteria_3<Tr> . . . page 3205
CGAL::Mesh_facet_criteria_3<Tr> . . . page 3203
CGAL::Mesh_facet_topology . . . page 3242

CGAL::Mesh_cell_criteria_3<Tr>

Definition

The class *Mesh_cell_criteria_3<Tr>* is a model of *MeshCellCriteria_3*. It provides, for the mesh tetrahedra, a uniform shape criteria and a sizing field which may be a uniform or variable field.

#include <CGAL/Mesh_cell_criteria_3.h>

Parameters

The parameter *Tr* must be identical to the nested type *Triangulation* of the instance used as model of *MeshComplex_3InTriangulation_3*.

Is Model for the Concepts

MeshCellCriteria_3

Types

typedef Tr::FT	*FT;*	Numerical type

Creation

Mesh_cell_criteria_3<Tr> mcc(FT radius_edge_bound, FT radius_bound);

Returns an object to serve as default criteria for cells. The argument *radius_edge_bound* is the upper bound for the radius-edge ratio of the tetrahedra. The argument *radius_bound* is a uniform upper bound for the circumradii of the tetrahedra in the mesh. See section 51.1 for further details. Note that if one parameter is set to 0, then its corresponding criteria is ignored.

template<class SizingField>
Mesh_cell_criteria_3<Tr> mcc(FT radius_edge_bound, SizingField radius_bound);

Returns an object to serve as default criteria for facets. The type *SizingField* must be a model of the concept *MeshDomainField_3*. The behavior and semantic of the arguments are the same as above, except that the radius bound parameter is a functional instead of a constant.

See Also

MeshCriteria_3 . . . . page 3175
MeshCellCriteria_3 . . . . page 3177
Mesh_criteria_3<Tr> . . . . page **??**
CGAL::make_mesh_3 . . . . page 3217

CGAL::Mesh_facet_criteria_3<Tr>

Functor
Class

Definition

The class *Mesh_facet_criteria_3<Tr>* is a model of *MeshFacetCriteria_3*. It provides a uniform bound for the shape criterion, a uniform or variable sizing field for the size criterion and/or a uniform or variable distance field for the approximation error criterion.

#include <CGAL/Mesh_facet_criteria_3.h>

Parameters

The parameter *Tr* must be identical to the nested type *Triangulation* of the instance used as model of *MeshComplex_3InTriangulation_3*.

Is Model for the Concepts

MeshFacetCriteria_3

Types

typedef Tr::Geom_traits::FT

FT;	Numerical type

Creation

Mesh_facet_criteria_3<Tr> mfc(FT angle_bound,
FT radius_bound,
FT distance_bound,
Mesh_facet_topology topology = FACET_VERTICES_ON_SURFACE)

Returns an object to serve as criteria for facets. The argument *angle_bound* is the lower bound for the angle in degrees of the surface mesh facets. The argument *radius_bound* is a uniform upper bound for the radius of the surface Delaunay balls. The argument *distance_bound* is an upper bound for the center-center distances of the surface mesh facets. *topology* is the set of topological constraints which have to be verified by each surface facet. See section 51.1 for further details. Note that if one parameter is set to 0, then its corresponding criteria is ignored.

template <class SizingField, class DistanceField>
Mesh_facet_criteria_3<Tr> mfc(FT angle_bound,
SizingField radius_bound,
DistanceField distance_bound,

Mesh_facet_topology topology = FACET_VERTICES_ON_SURFACE)

Returns an object to serve as criteria for facets. The types *SizingField* and *DistanceField* must be models of the concept *MeshDomainField_3*. The behavior and semantic of the arguments are the same as above, except that the radius and distance bound parameters are functionals instead of constants.

See Also

CGAL::Mesh_facet_topology . . . page 3242
MeshCriteria_3 . . . page 3175
MeshFacetCriteria_3 . . . page 3178
Mesh_criteria_3<Tr> . . . page **??**
MeshDomainField_3 . . . page 3191
CGAL::make_mesh_3 . . . page 3217

CGAL::Mesh_edge_criteria_3<Tr>

Functor
Class

Definition

The function object class *Mesh_edge_criteria_3<Tr>* is a model of *MeshEdgeCriteria_3*. It provides a bound for the size criterion.

#include <CGAL/Mesh_edge_criteria_3.h>

Is Model for the Concepts

MeshEdgeCriteria_3

Types

typedef Tr::Geom_traits::FT

FT;	Numerical type.

Creation

Mesh_edge_criteria_3<Tr> mec(FT length_bound);

Returns an object to serve as criteria for edges. The argument *length_bound* is an upper bound for the length of the edges which are used to discretize the curve segments. Note that if one parameter is set to 0, then its corresponding criteria is ignored.

template <class SizingField>
Mesh_edge_criteria_3<Tr> mec(SizingField length_bound);

Returns an object to serve as criteria for edges. The type *SizingField* must be a model of concept *MeshDomainField_3*. The behavior and semantic of the argument are the same as above, except that the length parameter is a functional instead of a constant.

See Also

MeshEdgeCriteria_3 . . . . page 3180
MeshCriteria_3 . . . . page 3175
Mesh_criteria_3<Tr> . . . . page **??**
MeshDomainField_3 . . . . page 3191

Functor
Class

CGAL::Mesh_constant_domain_field_3<Gt,Index>

Definition

The class *Mesh_constant_domain_field_3<Gt,Index>* is a model of concept *MeshDomainField_3*. It provides a constant field accessible using queries on 3D-points.

The class *Mesh_constant_domain_field_3<Gt,Index>* can also be customized through *set_size()* operations to become a piecewise constant field, i.e. a sizing field with a constant size on each subpart of the domain.

#include <CGAL/Mesh_constant_domain_field_3.h>

Parameters

The parameter *Gt* is the geometric traits class. It must match the type *Triangulation::Geom_traits*, where *Triangulation* is the nested type of the model of *MeshComplex_3InTriangulation_3* used in the meshing process.

The parameter *Index* is the type of index of the vertices of the triangulation. It must match the type *Index* of the model of *MeshDomain_3* used in the meshing process.

Is Model for the Concepts

MeshDomainField_3

Types

typedef Gt::FT	*FT;*	Numerical type.
typedef Gt::Point_3	*Point_3;*	Point type.
typedef Index	*Index;*	Type of index of the vertices of the triangulation.

Creation

Mesh_constant_domain_field_3<Gt,Index> mcf(FT size);

Builds a constant domain field with size *size*.

Operations

void	*mcf.set_size(FT size, int dimension, Index index)*

Sets the size such as *operator()* will return size *size* at any query point of dimension *dimension* and index *index*.

See Also

MeshDomainField_3 . . . . page 3191

CGAL::Implicit_mesh_domain_3<Function,BGT>

Class

Definition

The class *Implicit_mesh_domain_3<Function,BGT>* implements a domain whose bounding surface is described implicitly as the zero level set of a function. The domain to be discretized is assumed to be the domain where the function has negative values. This class is a model of the concept *MeshDomain_3*.

Parameters

Parameter *BGT* is a geometric traits which provides the basic operations to implement intersection tests and computations through a bisection method. This parameter must be instantiated with a model of the concept *BisectionGeometricTraits_3*.

Parameter *Function* provides the definition of the function. This parameter stands for a model of the concept *ImplicitFunction* described in the surface mesh generation package. The number types *Function::FT* and *BGT::FT* are required to match.

The constructor of *Implicit_mesh_domain_3<Function,BGT>* takes as argument a bounding sphere which is required to circumscribe the surface and to have its center inside the domain. This domain constructs intersection points between the surface and segments/rays/lines by bisection. It needs an *error_bound* such that the bisection process is stopped when the query segment is smaller than the error bound. The *error_bound* passed as argument to the domain constructor is a relative error bound expressed as a ratio to the bounding sphere radius.

#include <CGAL/Implicit_mesh_domain_3.h>

Is Model for the Concepts

MeshDomain_3

Creation

Implicit_mesh_domain_3<Function,BGT> domain(Function f, BGT::Sphere_3 bounding_sphere, BGT::FT error_bound = FT(1e-3))

f is the object of type *Function* that represents the implicit surface.
bounding_sphere is a bounding sphere of the implicit surface. The value of f at the sphere center c must be negative: $f(c) < 0$.
error_bound is the relative error bound used to compute intersection points between the implicit surface and query segments. The bisection is stopped when the length of the intersected segment is less than the product of *error_bound* by the radius of *bounding_sphere*.

See Also

MeshDomain_3 . . . . page 3168
BisectionGeometricTraits_3 . . . . page 3181
CGAL::make_mesh_3 . . . . page 3217.

CGAL::Polyhedral_mesh_domain_3<Polyhedron,IGT,TriangleAccessor>

Class

Definition

The class *Polyhedral_mesh_domain_3<Polyhedron,IGT,TriangleAccessor>* implements a domain whose boundary is a simplicial polyhedral surface. This surface must be closed and free of intersection. This class is a model of the concept *MeshDomain_3*.

Parameters

The parameter *Polyhedron* stands for the type of the input polyhedral surface. The only requirements for this type is that the triangles of the surfaces must be accessible through an object of the class *TriangleAccessor*.

The parameter *IGT* stands for a geometric traits class providing the types and functors required to implement the intersection tests and intersection computations for polyhedral boundary surfaces. This parameter has to be instantiated with a model of the concept *IntersectionGeometricTraits_3*.

The parameter *TriangleAccessor* provides access to the triangles of the input polyhedral surface. It must be a model of the concept *TriangleAccessor_3*. It defaults to *Triangle_accessor_3<Polyhedron,IGT>*. The type *IGT::Triangle_3* must be identical to the type *TriangleAccessor::Triangle_3*.

#include <CGAL/Polyhedral_mesh_domain_3.h>

Is Model for the Concepts

MeshDomain_3

Creation

Polyhedral_mesh_domain_3<Polyhedron,IGT,TriangleAccessor> domain(Polyhedron polyhedron);

Construction from a polyhedral surface.

See Also

MeshDomain_3 . . . . page 3168
TriangleAccessor_3 . . . . page 3193
IntersectionGeometricTraits_3 . . . . page 3184
CGAL::Triangle_accessor_3<Polyhedron_3<K>,K> . . . . page **??**
CGAL::make_mesh_3 . . . . page 3217.

Class

CGAL::Polyhedral_mesh_domain_with_features_3<IGT>

Definition

The class *Polyhedral_mesh_domain_with_features_3<IGT>* implements a domain whose boundary is a simplicial polyhedral surface. This surface must be closed and free of intersection. It is a model of the concept *MeshDomainWithFeatures_3*. It also provides a member function to automatically detect sharp features from the input polyhedral surface.

Parameters

The parameter *IGT* stands for a geometric traits class providing the types and functors required to implement the intersection tests and intersection computations for polyhedral boundary surfaces. This parameter has to be instantiated with a model of the concept *IntersectionGeometricTraits_3*.

#include <CGAL/Polyhedral_mesh_domain_with_features_3.h>

Is Model for the Concepts

MeshDomainWithFeatures_3

Inherits From

CGAL::Mesh_domain_with_polyline_features_3<CGAL::Polyhedral_mesh_domain_3<CGAL::Mesh_polyhedron_3<IGT>::type,IGT> >

Types

Polyhedral_mesh_domain_with_features_3<IGT>:: FT

Numerical type.

Creation

template <typename Polyhedron>
Polyhedral_mesh_domain_with_features_3<IGT> md(Polyhedron p);

Constructs a *Polyhedral_mesh_domain_with_features_3* from a *Polyhedron*. The only requirement on type *Polyhedron* is that *CGAL::Mesh_polyhedron_3<IGT>::type* should be constructible from *Polyhedron*. No feature detection is done at this level. Note that a copy of *p* will be done.

Polyhedral_mesh_domain_with_features_3<IGT> md(std::string filename);

Constructs a *Polyhedral_mesh_domain_with_features_3* from an off file. No feature detection is done at this level.

Operations

void *md.detect_features(FT angle_bound=120)*

Detects sharp features of the internal polyhedron and inserts them in as features of the domain. *angle_bound* gives the maximum dihedral angle (in degrees) between two triangles of the internal polyhedron which is used to consider that the edge between those triangles is a feature edge.

See Also

MeshDomainWithFeatures_3 . . . . page 3171
CGAL::Mesh_domain_with_polyline_features_3<MeshDomain> . . . . page **??**
CGAL::Polyhedral_mesh_domain_3<Polyhedron,IGT,TriangleAccessor> . . . . page 3209
CGAL::Mesh_polyhedron_3<IGT> . . . . page 3215

Class

CGAL::Labeled_image_mesh_domain_3<Image,BGT>

Definition

The class *Labeled_image_mesh_domain_3<Image,BGT>* implements a domain described by a 3D labeled image. A 3D labeled image is a grid of voxels, where each voxel is associated with an index (a subdomain index) characterizing the subdomain in which the voxel lies. This class is a model of the concept *MeshDomain_3*. The domain to be discretized is the union of voxels that have an non-default index (different from the default constructed value of the type *Image::Type*).

This class includes a member function that provides, by interpolation, the subdomain index of any query point. An intersection between a segment and bounding surfaces is detected when both segment endpoints are associated with different values of subdomain indices. The intersection is then constructed by bisection. The bisection stops when the query segment is shorter than a given error bound e. This error bound is given by $e{=}d{\times}bound$ where d is the length of the diagonal of the bounding box (in world coordinates) and *bound* is the argument passed to the constructor of *Labeled_image_mesh_domain_3<Image,BGT>*.

Parameters

Parameter *Image* is the type of the input image. This parameter must be a model of the concept *LabeledImage_3*.

Parameter *BGT* is a geometric traits class which provides the basic operations to implement intersection tests and intersection computations through a bisection method. This parameter must be instantiated with a model of the concept *BisectionGeometricTraits_3*.

#include <CGAL/Labeled_image_mesh_domain_3.h>

Is Model for the Concepts

MeshDomain_3

Creation

Labeled_image_mesh_domain_3<Image,BGT> domain(Image image, BGT::FT error_bound = FT(1e-3));

Construction from an image. The parameter *error_bound* is relative to the size of the image.

See Also

MeshDomain_3 . . . . page 3168
BisectionGeometricTraits_3 . . . . page 3181
CGAL::make_mesh_3 . . . . page 3217.

CGAL::Mesh_domain_with_polyline_features_3<MeshDomain_3>

Class

Definition

The class *Mesh_domain_with_polyline_features_3<MeshDomain_3>* is designed to allow the user to add some 0- and 1-dimensional features into any model of the *MeshDomain_3* concept. The 1-dimensional features are described as polylines whose endpoints are the added corners.

Parameters

The template parameter *MeshDomain_3* is the type of the domain which should be extended. It has to be a model of the *MeshDomain_3* concept.

#include <CGAL/Mesh_domain_with_polyline_features_3.h>

Is Model for the Concepts

MeshDomainWithFeatures_3

Inherits From

MeshDomain_3

Types

typedef int	*Corner_index;*	*Corner_index* type.
typedef int	*Curve_segment_index;*	*Curve_segment_index* type.

Creation

template <typename ...T>
Mesh_domain_with_polyline_features_3<MeshDomain_3> md(T ...t);

Constructor. Forwards the arguments to the constructor of the base class.

Operations

template <typename InputIterator>
void *md.add_features(InputIterator begin, InputIterator beyond)*

Add 1-dimensional features in the domain. *InputIterator* value type must be a model of the concept *MeshPolyline_3*.

template <typename InputIterator>
void *md.add_features_and_incidences(InputIterator begin, InputIterator beyond)*

Add 1-dimensional features in the domain with their incidences with 2-dimensional features of the domain. The *InputIterator* value type must be *std::pair<Polyline, std::pair< InputSurfacePatchIndexIterator, InputSurfacePatchIndexIterator>* > where *Polyline* must be a model of the concept *MeshPolyline_3* and the internal pair gives a range on surface patches indices which are incident to the polyline.

See Also

MeshDomain_3 . . . . page 3168
MeshDomainWithFeatures_3 . . . . page 3171
MeshPolyline_3 . . . . page 3192
CGAL::Implicit_mesh_domain_3<Function,BGT> . . . . page 3207
CGAL::Polyhedral_mesh_domain_3<Polyhedron,IGT,TriangleAccessor> . . . . page 3209
CGAL::Labeled_image_mesh_domain_3<Image,BGT> . . . . page 3212

CGAL::Mesh_polyhedron_3<IGT>

Class

Definition

The class *Mesh_polyhedron_3<IGT>* provides a customized *CGAL::Polyhedron_3* type. This type uses as *PolyhedronItems_3* a customized type which adds data to the Vertex, Face and Halfedge class. Those data are required to use our sharp features detection algorithm.

#include <CGAL/Mesh_polyhedron_3.h>

Parameters

Template parameter *IGT* stands for the geometric traits associated to the meshing process. It should be a model of the two concepts *PolyhedronTraits_3* and *IntersectionGeometricTraits_3*.

Types

Mesh_polyhedron_3<IGT>:: type	*CGAL::Polyhedron_3<IGT>* type with customized *PolyhedronItems_3* designed to handle sharp feature detection.

See Also

CGAL::Polyhedron_3<Gt> . . . page **??** *CGAL::Polyhedral_mesh_domain_with_features_3<IGT>* . . . page 3210

Class

CGAL::Triangle_accessor_3<CGAL::Polyhedron_3<K>,K>

Definition

The class *Triangle_accessor_3<CGAL::Polyhedron_3<K>,K>* is a model for the concept *TriangleAccessor_3*. It is designed to serve as accessor for objects of type *CGAL::Polyhedron_3<K>*.

advanced

Actually, the class *Triangle_accessor_3<CGAL::Polyhedron_3<K>,K>* is a partial specialization of the class template *template<typename Polyhedron, typename K> Triangle_accessor_3<Polyhedron, K>*. One may give another partial specialization of this class to handle one's own polyhedron data structure.

advanced

#include <CGAL/Triangle_accessor_3.h>

Parameters

The template parameter *K* is the geometric traits class.

Is Model for the Concepts

TriangleAccessor_3

Types

typedef Polyhedron_3<K>::Facet_const_iterator

Triangle_iterator;	Triangle iterator.

typedef Polyhedron_3<K>::Facet_const_handle

Triangle_handle;	Triangle handle.

typedef K::Triangle_3

Triangle_3;	Triangle type.

See Also

TriangleAccessor_3 . . . . page 3193
CGAL::Polyhedral_mesh_domain_3<Polyhedron,IGT,TriangleAccessor> . . . . page 3209

CGAL::make_mesh_3

Definition

The function *make_mesh_3* is a 3D mesh generator. It produces simplicial meshes which discretize 3D domains.

The mesh generation algorithm is a Delaunay refinement process followed by an optimization phase. The criteria driving the Delaunay refinement process may be tuned to achieve the user needs with respect to the size of mesh elements, the accuracy of boundaries approximation, etc.

The optimization phase is a sequence of optimization processes, including possibly a Lloyd smoothing, an odt-smoothing, a perturber and an exuder. Each optimization process can be activated or not, according to the user requirements and available time. By default, only the perturber and the exuder are activated. Note that the benefits of the exuder will be lost if the mesh is further refined afterward.

The function outputs the mesh to an object which provides iterators to traverse the resulting mesh data structure or can be written to a file (see 51.3).

```
template <class C3T3, class MeshDomain_3, class MeshCriteria>
C3T3    make_mesh_3( MeshDomain_3 domain,
                     MeshCriteria criteria,
                     parameters::internal::Features_options features = parameters::features(domain),
                     parameters::internal::Lloyd_options lloyd = parameters::no_lloyd(),
                     parameters::internal::Odt_options odt = parameters::no_odt(),
                     parameters::internal::Perturb_options perturb = parameters::perturb(),
                     parameters::internal::Exude_options exude = parameters::exude())
```

Parameters

Parameter *C3T3* is required to be a model of the concept *MeshComplex_3InTriangulation_3*. This is the return type.

The type *C3T3* is in particular required to provide a nested type *C3T3::Triangulation* for the 3D triangulation embedding the mesh. The vertex and cell base classes of the triangulation *C3T3::Triangulation* are required to be models of the concepts *MeshVertexBase_3* and *MeshCellBase_3* respectively.

Template parameter *MeshDomain_3* is required to be a model of the concept *MeshDomain_3*, or of the refined concept *MeshDomainWithFeatures_3* if the domain has corners and curve segments that need to be accurately represented in the mesh. The argument *domain* is the sole link through which the domain to be discretized is known by the mesh generation algorithm.

The parameter *features* allows the user to specify if 0 and 1-dimensional features actually have to be taken into account or not when the domain is a model of *MeshDomainWithFeatures_3*. The type *Features* of this parameter is an internal undescribed type. The library provides functions to construct appropriate values of that type.

- *parameters::features(domain)* sets *features* according to the domain, i.e. 0 and1-dimensional features are taken into account if *domain* is a *MeshDomainWithFeatures_3*. This is the default behavior if parameter *features* is not specified.

- *parameters::no_features()* prevents the representation of 0 and 1-dimensional features in the mesh.

The template parameter *MeshCriteria* has to be a model of the concept *MeshCriteria_3*, or a model of the refined concept *MeshCriteriaWithFeatures_3* if the domain has exposed features. The argument *criteria* of type *MeshCriteria* specifies the size and shape requirements for mesh tetrahedra and surface facets. These criteria form the rules which drive the refinement process. All mesh elements satisfy those criteria at the end of the refinement process. In addition, if the domain has features, the argument *criteria* provides a sizing field to guide the discretization of 1-dimensional exposed features.

The four additional parameters are optimization parameters. They control which optimization processes are performed and allow the user to tune the parameters of the optimization processes. We do not describe the types of optimization parameters as they are internal types. The package defines two global functions for each optimization parameter to generate appropriate value of this parameter.

- *parameters::lloyd()* and *parameters::no_lloyd()* are designed to trigger or not a call to *lloyd_optimize_mesh_3* function and to set the parameters of this optimizer. If one parameter is not set, the default value of *lloyd_optimize_mesh_3* is used for this parameter.
- *parameters::odt()* and *parameters::no_odt()* are designed to trigger or not a call to *odt_optimize_mesh_3* function and to set the parameters of this optimizer If one parameter is not set, the default value of *odt_optimize_mesh_3* is used for this parameter.
- *parameters::perturb()* and *parameters::no_perturb()* are designed to trigger or not a call to *perturb_mesh_3* function and to set the parameters of this optimizer. If one parameter is not set, the default value of *perturb_mesh_3* is used for this parameter, except for the time bound which is set to be equal to the refinement CPU time.
- *parameters::exude()* and *parameters::no_exude()* are designed to trigger or not a call to *exude_mesh_3* function and to override to set the parameters of this optimizer. If one parameter is not set, the default value of *exude_mesh_3* is used for this parameter, except for the time bound which is set to be equal to the refinement CPU time.

The optimization parameters can be passed in random order. If one parameter is not passed, its default value is used. The default values are *no_lloyd()*, *no_odt()*, *perturb()* and *exude()*.

Note that whatever may be the optimization processes activated, they are always launched in the order that is a suborder of the following (see user manual for further details): *lloyd*, *odt*, *perturb*, *exude*.

Also beware, that optimization of the mesh is obtained by perturbing mesh vertices and modifying the mesh connectivity and that this has an impact on the strict compliance to the refinement criteria. Though a strict compliance to mesh criteria is granted at the end of the Delaunay refinement, this may no longer be true after some optimization processes. Also beware that the default behavior does involve some optimization processes.

See Also

refine_mesh_3 . . . . page **??**
parameters::features . . . . page **??**
parameters::no_features . . . . page **??**
exude_mesh_3 . . . . page **??**
perturb_mesh_3 . . . . page **??**
lloyd_optimize_mesh_3 . . . . page **??**
odt_optimize_mesh_3 . . . . page **??**
parameters::exude . . . . page **??**

Function

CGAL::refine_mesh_3

Definition

The function *refine_mesh_3* is a 3D mesh generator. It produces simplicial meshes which discretize 3D domains.

The mesh generation algorithm is a Delaunay refinement process followed by an optimization phase. The criteria driving the Delaunay refinement process may be tuned to achieve the user needs with respect to the size of mesh elements, the accuracy of boundaries approximation, etc.

The optimization phase is a succession of optimization processes, including possibly a Lloyd smoothing, an odt-smoothing, a perturber and an exuder. Each optimization process can be activated or not, according to the user requirements and available time. By default, only the perturber and the exuder are activated. Note that the benefits of the exuder will be lost if the mesh is further refined afterward.

template <class C3T3, class MeshDomain_3, class MeshCriteria>
void refine_mesh_3(C3T3& c3t3,
MeshDomain_3 mesh_domain,
MeshCriteria mesh_criteria,
parameters::internal::Lloyd_options lloyd = parameters::no_lloyd(),
parameters::internal::Odt_options odt = parameters::no_odt(),
parameters::internal::Perturb_options perturb = parameters::perturb(),
parameters::internal::Exude_options exude = parameters::exude())

advanced

The function template *refine_mesh_3* may be used to refine a previously computed mesh, e.g.:

C3T3 c3t3 = CGAL::make_mesh_3<C3T3>(domain,criteria);
CGAL::refine_mesh_3(c3t3, domain, new_criteria);

Please note that we guarantee the result if and only if the domain does not change from one refinement to the next one.

advanced

Parameters

Parameter *C3T3* is required to be a model of the concept *MeshComplex_3InTriangulation_3*. The argument *c3t3* is passed by reference as this object is modified by the refinement process. As the refinement process only adds points to the triangulation, all vertices of the triangulation of *c3t3* remain in the mesh during the refinement process. Object *c3t3* can be used to insert specific points in the domain to ensure that they will be contained in the final triangulation.

The type *C3T3* is in particular required to provide a nested type *C3T3::Triangulation* for the 3D triangulation embedding the mesh. The vertex and cell base classes of the triangulation *C3T3::Triangulation* are required to be models of the concepts *MeshVertexBase_3* and *MeshCellBase_3* respectively.

Template parameter *MeshDomain_3* is required to be a model of the concept *MeshDomain_3* or of the refined concept *MeshDomainWithFeatures_3* if 0 and 1-dimensional features of the input complex have to be accurately represented in the mesh. The argument *domain* is the sole link through which the domain to be discretized is known by the mesh generation algorithm.

The template parameter *MeshCriteria* has to be a model of the concept *MeshCriteria_3*, or a model of the refined concept *MeshCriteriaWithFeatures_3* if the domain has exposed features. The argument *criteria* of type *MeshCriteria* specifies the size and shape requirements for mesh tetrahedra and surface facets. These criteria form the rules which drive the refinement process. All mesh elements satisfy those criteria at the end of the refinement process. In addition, if the domain has features, the argument *criteria* provides a sizing field to guide the discretization of 1-dimensional exposed features.

The four additional parameters are optimization parameters. They control which optimization processes are performed and allow the user to tune the parameters of the optimization processes. We do not describe the types of optimization parameters as they are internal types. The package defines two global functions for each optimization parameter to generate appropriate value of this parameter.

- *parameters::lloyd()* and *parameters::no_lloyd()* are designed to trigger or not a call to *lloyd_optimize_mesh_3* function and to set the parameters of this optimizer. If one parameter is not set, the default value of *lloyd_optimize_mesh_3* is used for this parameter.
- *parameters::odt()* and *parameters::no_odt()* are designed to trigger or not a call to *odt_optimize_mesh_3* function and to set the parameters of this optimizer If one parameter is not set, the default value of *odt_optimize_mesh_3* is used for this parameter.
- *parameters::perturb()* and *parameters::no_perturb()* are designed to trigger or not a call to *perturb_mesh_3* function and to set the parameters of this optimizer. If one parameter is not set, the default value of *perturb_mesh_3* is used for this parameter, except for the time bound which is set to be equal to the refinement CPU time.
- *parameters::exude()* and *parameters::no_exude()* are designed to trigger or not a call to *exude_mesh_3* function and to override to set the parameters of this optimizer. If one parameter is not set, the default value of *exude_mesh_3* is used for this parameter, except for the time bound which is set to be equal to the refinement CPU time.

The optimization parameters can be passed in random order. If one parameter is not passed, its default value is used. The default values are *no_lloyd()*, *no_odt()*, *perturb()* and *exude()*. Note that whatever may be the optimization processes activated, they are always launched in the order that is a suborder of the following (see user manual for further details): *lloyd*, *odt*, *perturb*, *exude*.

Beware that optimization of the mesh is obtained by perturbing mesh vertices and modifying the mesh connectivity and that this has an impact on the strict compliance to the refinement criteria. Though a strict compliance to mesh criteria is granted at the end of the Delaunay refinement, this may no longer be true after some optimization processes. Also beware that the default behavior does involve some optimization processes.

See Also

CGAL::make_mesh_3 . . . . . . page 3217
CGAL::exude_mesh_3 . . . . . . page 3223
CGAL::perturb_mesh_3 . . . . . . page 3225
CGAL::lloyd_optimize_mesh_3 . . . . . . page 3227
CGAL::odt_optimize_mesh_3 . . . . . . page 3229
CGAL::parameters::exude . . . . . . page 3233
CGAL::parameters::no_exude . . . . . . page 3234
CGAL::parameters::perturb . . . . . . page 3235
CGAL::parameters::no_perturb . . . . . . page 3236

CGAL::exude_mesh_3

Function

Definition

The function *exude_mesh_3* performs a sliver exudation process on a Delaunay mesh.

The sliver exudation process consists in turning the Delaunay triangulation into a weighted Delaunay triangulation and optimizing the weights of vertices in such a way that slivers disappear and the quality of the mesh improves.

#include <CGAL/exude_mesh_3.h>

template<typename C3T3>
Mesh_optimization_return_code

exude_mesh_3(C3T3& c3t3,
double parameters::time_limit=0,
double parameters::sliver_bound=0)

Precondition: *time_limit* ≥ 0 and $0 \leq$ *sliver_bound* ≤ 180

Parameters

Parameter *C3T3* is required to be a model of the concept *MeshComplex_3InTriangulation_3*. The argument *c3t3*, passed by reference, provides the initial mesh and is modified by the algorithm to represent the final optimized mesh.

The function has two optional parameters which are named parameters (we use the Boost.Parameter library). Therefore, when calling the function, the parameters can be provided in any order provided that the names of the parameters are used (see example at the bottom of this page).

- Parameter *time_limit*, whose name is *parameters::time_limit*, is used to set up, in seconds, a CPU time limit after which the optimization process is stopped. This time is measured using the *CGAL::Timer* class. The default value is *0* and means that there is no time limit.
- Parameter *sliver_bound*, whose name is *parameters::sliver_bound*, is designed to give, in degree, a targeted lower bound on dihedral angles of mesh cells. The exudation process considers in turn all the mesh cells that have a smallest dihedral angle less than *sliver_bound* and tries to make them disappear by weighting their vertices. The optimization process stops when every cell in the mesh achieves this quality. The default value is *0* and means that there is no targeted bound : the exuder then runs as long as it can improve the smallest dihedral angles of the set of cells incident to some vertices.

Return Values

The function *exude_mesh_3* returns a value of type *Mesh_optimization_return_code* which is:

- *BOUND_REACHED* when the targeted bound for the smallest dihedral angle in the mesh is reached.

- *TIME_LIMIT_REACHED* when the time limit is reached.
- *CANT_IMPROVE_ANYMORE* when exudation process stops because it can no longer improve the smallest dihedral angle of the set of cells incident to some vertex in the mesh.

Example

```
// Exude without sliver_bound, using at most 10s CPU time
exude_mesh_3(c3t3, parameters::time_limit=10);
```

See Also

CGAL::Mesh_optimization_return_code . . . . . . page 3241
CGAL::make_mesh_3 . . . . . . page 3217
CGAL::refine_mesh_3 . . . . . . page 3220
CGAL::perturb_mesh_3 . . . . . . page 3225
CGAL::lloyd_optimize_mesh_3 . . . . . . page 3227
CGAL::odt_optimize_mesh_3 . . . . . . page 3229

CGAL::perturb_mesh_3

Definition

The function *perturb_mesh_3* is a mesh optimizer that improves the quality of a Delaunay mesh by changing the mesh vertices positions.

The perturber tries to improve the dihedral angles of the worst cells in the mesh degree by degree: the step number n is considered as successful if after this step the worst tetrahedron of the mesh has a minimal dihedral angle larger than n degrees. The perturber exits if this is not the case.

#include <CGAL/perturb_mesh_3.h>

```
template<typename C3T3, typename MeshDomain_3>
Mesh_optimization_return_code

                perturb_mesh_3( C3T3& c3t3,
                                MeshDomain_3 domain,
                                double parameters::time_limit=0,
                                double parameters::sliver_bound=0)
```

Precondition: $time\_limit \geq 0$ and $0 \leq sliver\_bound \leq 180$

Parameters

Parameter *C3T3* is required to be a model of the concept *MeshComplex_3InTriangulation_3*. The argument *c3t3*, passed by reference, provides the initial mesh and is modified by the algorithm to represent the final optimized mesh.

Parameter *MeshDomain_3* is required to be a model of the concept *MeshDomain_3*. The argument *domain* must be the *MeshDomain_3* object used to create the *c3t3* parameter.

The function has two optional parameters which are named parameters (we use the Boost.Parameter library). Therefore, when calling the function, the parameters can be provided in any order provided that the names of the parameters are used (see example at the bottom of this page).

- Parameter *time_limit* whose name is *parameters::time_limit*, is used to set up, in seconds, a CPU time limit after which the optimization process is stopped. This time is measured using *CGAL::Timer*. The default value is *0* and means that there is no time limit.
- Parameter *sliver_bound*, whose name is *parameters::sliver_bound*, is designed to give, in degree, a targeted lower bound on dihedral angles of mesh cells. The function *perturb_mesh_3* runs as long as steps are successful and step number *sliver_bound* (after which the worst tetrahedron in the mesh has a smallest angle larger than *sliver_bound* degrees) has not been reached. The default value is *0* and means that there is no targeted bound: the perturber then runs as long as steps are successful.

Return Values

The function *perturb_mesh_3* returns a value of type *Mesh_optimization_return_code* which is:

- *BOUND_REACHED* when the targeted bound for the smallest dihedral angle in the mesh is reached.
- *TIME_LIMIT_REACHED* when the time limit is reached.
- *CANT_IMPROVE_ANYMORE* when the perturbation process stops because the last step is unsuccessful.

Example

```
// Perturb until every dihedral angle of the mesh is >= 10 degrees
// No time bound is set
perturb_mesh_3(c3t3, domain, parameters::sliver_bound=10);
```

See Also

CGAL::Mesh_optimization_return_code . . . . page 3241
CGAL::make_mesh_3 . . . . page 3217
CGAL::refine_mesh_3 . . . . page 3220
CGAL::exude_mesh_3 . . . . page 3223
CGAL::lloyd_optimize_mesh_3 . . . . page 3227
CGAL::odt_optimize_mesh_3 . . . . page 3229

CGAL::lloyd_optimize_mesh_3

Definition

The function *lloyd_optimize_mesh_3* is a mesh optimization process based on the minimization of a global energy function.

In *lloyd_optimize_mesh_3*, the minimized global energy may be interpreted as the L^1-norm of the error achieved when the function x^2 is interpolated on the mesh domain using a piecewise linear function which is linear in each cell of the Voronoi diagram of the mesh vertices.

The optimizer *lloyd_optimize_mesh_3* works in iterative steps. At each iteration, mesh vertices are moved into positions that bring to zero the energy gradient and the Delaunay triangulation is updated. Vertices on the mesh boundaries are handled in a special way so as to preserve an accurate representation of the domain boundaries.

#include <CGAL/lloyd_optimize_mesh_3.h>

```
template<typename C3T3, typename MeshDomain_3>
Mesh_optimization_return_code

        lloyd_optimize_mesh_3( C3T3& c3t3,
                               MeshDomain_3 domain,
                               double parameters::time_limit=0,
                               std::size_t parameters::max_iteration_number=0,
                               double parameters::convergence=0.02,
                               double parameters::freeze_bound = 0.01)
```

Precondition: *time_limit* ≥ 0 and $0 \leq$ *convergence* ≤ 1 and $0 \leq$ *freeze_bound* ≤ 1

Parameters

Parameter *C3T3* is required to be a model of the concept *MeshComplex_3InTriangulation_3*. The argument *c3t3*, passed by reference, provides the initial mesh and is modified by the algorithm to represent the final optimized mesh.

Parameter *MeshDomain_3* is required to be a model of the concept *MeshDomain_3*. The argument *domain* must be the *MeshDomain_3* object used to create the *c3t3* parameter.

The function has four optional parameters which are named parameters (we use the Boost.Parameter library). Therefore, when calling the function, the parameters can be provided in any order provided that the names of the parameters are used (see example at the bottom of this page).

- Parameter *time_limit*, whose name is *parameters::time_limit*, is used to set up, in seconds, a CPU time limit after which the optimization process is stopped. This time is measured using *CGAL::Timer*. The default value is *0* and means that there is no time limit.
- Parameter *max_iteration_number*, whose name is *parameters::max_iteration_number*, sets a limit on the number of performed iterations. The default value of *0* means that there is no limit on the number of performed iterations.

- Parameter *convergence*, whose name is *parameters::convergence*, is a stopping criterion based on convergence: the optimization process is stopped, when at the last iteration, the displacement of any vertex is less than a given percentage of the length of the shortest edge incident to that vertex. The parameter *convergence* gives the threshold ratio.
- Parameter *freeze_bound*, whose name is *parameters::freeze_bound*, is designed to reduce running time of each optimization iteration. Any vertex that has a displacement less than a given percentage of the length of its shortest incident edge, is frozen (i.e. is not relocated). The parameter *freeze_bound* gives the threshold ratio. At each iteration, any vertex that moves, unfreezes all its incident vertices.

Return Values

The function *lloyd_optimize_mesh_3* returns a value of type *Mesh_optimization_return_code* which is:

- *TIME_LIMIT_REACHED* when the time limit is reached.
- *MAX_ITERATION_NUMBER_REACHED* when *lloyd_optimize_mesh_3* stops because it has performed *max_iteration_number* iterations.
- *CONVERGENCE_REACHED* when *lloyd_optimize_mesh_3* stops because the convergence criterion is achieved.

Example

```
// Lloyd-smoothing until convergence reaches 0.01, freezing vertices
   which
// move less than 0.001*shortest_incident_edge_length
lloyd_optimize_mesh_3(c3t3, domain, parameters::convergence=0.01,
                      parameters::freeze_bound=0.001);
```

See Also

CGAL::Mesh_optimization_return_code . . . . page 3241
CGAL::make_mesh_3 . . . . page 3217
CGAL::refine_mesh_3 . . . . page 3220
CGAL::exude_mesh_3 . . . . page 3223
CGAL::perturb_mesh_3 . . . . page 3225
CGAL::odt_optimize_mesh_3 . . . . page 3229

CGAL::odt_optimize_mesh_3

Function

Definition

The function *odt_optimize_mesh_3* is a mesh optimization process based on the minimization of a global energy function.

In *odt_optimize_mesh_3*, the minimized global energy may be interpreted as the L^1-norm of the error achieved when the function x^2 is interpolated on the mesh domain using a piecewise linear function which is linear in each mesh cell.

The optimizer *odt_optimize_mesh_3* works in iterative steps. At each iteration, mesh vertices are moved into positions that bring to zero the energy gradient and the Delaunay triangulation is updated. Vertices on the mesh boundaries are handled in a special way so as to preserve an accurate representation of the domain boundaries.

#include <CGAL/odt_optimize_mesh_3.h>

```
template<typename C3T3, typename MeshDomain_3>
Mesh_optimization_return_code

                    odt_optimize_mesh_3( C3T3& c3t3,
                                         MeshDomain_3 domain,
                                         double parameters::time_limit=0,
                                         std::size_t parameters::max_iteration_number=0,
                                         double parameters::convergence=0.02,
                                         double parameters::freeze_bound = 0.01)
```

Precondition: *time_limit* ≥ 0 and $0 \leq$ *convergence* ≤ 1 and $0 \leq$ *freeze_bound* ≤ 1

Parameters

Parameter *C3T3* is required to be a model of the concept *MeshComplex_3InTriangulation_3*. The argument *c3t3*, passed by reference, provides the initial mesh and is modified by the algorithm to represent the final optimized mesh.

Parameter *MeshDomain_3* is required to be a model of the concept *MeshDomain_3*. The argument *domain* must be the *MeshDomain_3* object used to create the *c3t3* parameter.

The function has four optional parameters which are named parameters (we use the Boost.Parameter library). Therefore, when calling the function, the parameters can be provided in any order provided that the names of the parameters are used (see example at the bottom of this page).

- Parameter *time_limit*, whose name is *parameters::time_limit*, is used to set up, in seconds, a CPU time limit after which the optimization process is stopped. This time is measured using *CGAL::Timer*. The default value is *0* and means that there is no time limit.
- Parameter *max_iteration_number*, whose name is *parameters::max_iteration_number*, sets a limit on the number of performed iterations. The default value of *0* means that there is no limit on the number of performed iterations.

- Parameter *convergence*, whose name is *parameters::convergence*, is a stopping criterion based on convergence: the optimization process is stopped, when at the last iteration, the displacement of any vertex is less than a given percentage of the length the shortest edge incident to that vertex. The parameter *convergence* gives the threshold ratio.

- Parameter *freeze_bound*, whose name is *parameters::freeze_bound*, is designed to reduce running time of each optimization iteration. Any vertex that has a displacement less than a given percentage of the length of its shortest incident edge, is frozen (i.e. is not relocated). The parameter *freeze_bound* gives the threshold ratio. At each iteration, any vertex that moves, unfreezes the neighboring vertices.

Return Values

The function *odt_optimize_mesh_3* returns a value of type *Mesh_optimization_return_code* which is:

- *TIME_LIMIT_REACHED* when the time limit is reached.

- *MAX_ITERATION_NUMBER_REACHED* when *odt_optimize_mesh_3* stops because it has performed *max_iteration_number* iterations.

- *CONVERGENCE_REACHED* when *odt_optimize_mesh_3* stops because the convergence criterion is achieved.

Example

```
// 100 iterations of Odt-smoothing
odt_optimize_mesh_3(c3t3, domain, parameters::max_iteration_number=100,
                         parameters::convergence=0);
```

See Also

CGAL::Mesh_optimization_return_code . . . . page 3241
CGAL::make_mesh_3 . . . . page 3217
CGAL::refine_mesh_3 . . . . page 3220
CGAL::exude_mesh_3 . . . . page 3223
CGAL::perturb_mesh_3 . . . . page 3225
CGAL::lloyd_optimize_mesh_3 . . . . page 3227

CGAL::parameters::features

Definition

The function *parameters::features* provides a value of internal type *Features* to specify if 0 and 1-dimensional features have to be taken into account. The provided value is a default value that triggers the representation of corners and curve segments in the mesh when the domain is a model of *MeshDomainWithFeatures_3*.

#include <CGAL/refine_mesh_3.h>

% parameters::internal::Features_options

parameters::features()

Provides a *Features_options* value such that 0 and 1-dimensional input features are taken into account if domain is a model of the refined concept *MeshDomainWithFeatures_3*.

See Also

CGAL::make_mesh_3 . . . . page 3217,
CGAL::refine_mesh_3 . . . . page 3220,
CGAL::parameters::no_features . . . . page 3232.

Function

CGAL::parameters::no_features

Definition

The function *parameters::no_features* allows the user to prevent the handling of 0 and 1 dimensional features. This is useful when the domain is a model of *MeshDomainWithFeatures_3* and the user does not want corners and curve segments to be accurately represented in the mesh.

#include <CGAL/refine_mesh_3.h>

parameters::internal::Features_options

parameters::no_features()

Return a *Features_options* value that prevents the mesh generator to take into account 0 and 1-dimensional input features.

See Also

CGAL::make_mesh_3 . . . . page 3217,
CGAL::refine_mesh_3 . . . . page 3220,
CGAL::parameters::features . . . . page 3231.

CGAL::parameters::exude

Function

Definition

The function *parameters::exude* allows the user to trigger a call to *exude_mesh_3* in the *make_mesh_3* and *refine_mesh_3* mesh generation functions. It also allows the user to pass parameters to the optimization function *exude_mesh_3* through these mesh generation functions.

#include <CGAL/refine_mesh_3.h>

parameters::internal::Exude_options

parameters::exude(double parameters::time_limit = 0,
double parameters::sliver_bound = 0)

Parameters

The parameters are named parameters. They are the same (i.e. they have the same name and the same default values) as the parameters of *exude_mesh_3* function. See its manual page for further details.

Example

```
// Mesh generation with an exudation step
C3t3 c3t3 = make_mesh_3<c3t3>(domain, criteria, parameters::exude());
refine_mesh_3(c3t3, domain, criteria,
    parameters::exude(parameters::time_limit=10));
```

See Also

CGAL::no_exude . . . . page **??**
CGAL::exude_mesh_3 . . . . page 3223
CGAL::make_mesh_3 . . . . page 3217
CGAL::refine_mesh_3 . . . . page 3220

Function

CGAL::parameters::no_exude

Definition

The function *parameters::no_exude* allows the user to tell the mesh generation functions *make_mesh_3* and *refine_mesh_3* that no exudation must be done.

#include <CGAL/refine_mesh_3.h>

parameters::internal::Exude_options

parameters::no_exude()

Example

```
// Mesh generation without exudation
C3t3 c3t3 = make_mesh_3<c3t3>(domain, criteria, parameters::no_exude());
```

See Also

CGAL::parameters::exude . . . . page 3233
CGAL::exude_mesh_3 . . . . page 3223
CGAL::make_mesh_3 . . . . page 3217
CGAL::refine_mesh_3 . . . . page 3220

CGAL::parameters::perturb

Definition

The function *parameters::perturb* allows the user to trigger a call to *perturb_mesh_3* in *make_mesh_3* and *refine_mesh_3* mesh generation functions. It also allows the user to pass parameters to the optimization function *perturb_mesh_3* through these mesh generation functions.

#include <CGAL/refine_mesh_3.h>

parameters::internal::Perturb_options

parameters::perturb(double parameters::time_limit = 0,
double parameters::sliver_bound = 0)

Parameters

The parameters are named parameters. They are the same (i.e. they have the same name and the same default values) as the parameters of *perturb_mesh_3* function. See its manual page for further details.

Example

```
// Mesh generation with a perturbation step
C3t3 c3t3 = make_mesh_3<c3t3>(domain, criteria, parameters::perturb());
refine_mesh_3(c3t3, domain, criteria,
   parameters::perturb(parameters::time_limit=10));
```

See Also

CGAL::no_perturb . . . . page **??**
CGAL::perturb_mesh_3 . . . . page 3225
CGAL::make_mesh_3 . . . . page 3217
CGAL::refine_mesh_3 . . . . page 3220

Function

Function

CGAL::parameters::no_perturb

Definition

The function *parameters::no_perturb* allows the user to tell mesh generation global functions *make_mesh_3* and *refine_mesh_3* that no perturbation must be done.

#include <CGAL/refine_mesh_3.h>

parameters::internal::Perturb_options

parameters::no_perturb()

Parameters

This function takes no parameter.

Example

```
// Mesh generation without perturbation
C3t3 c3t3 = make_mesh_3<c3t3>(domain, criteria,
    parameters::no_perturb());
```

See Also

CGAL::parameters::perturb . . . page 3235
CGAL::perturb_mesh_3 . . . page 3225
CGAL::make_mesh_3 . . . page 3217
CGAL::refine_mesh_3 . . . page 3220

CGAL::parameters::lloyd

Definition

The function *parameters::lloyd* allows the user to trigger a call of *lloyd_optimize_mesh_3* in the mesh generation functions *make_mesh_3* and *refine_mesh_3*. It also allows the user to pass parameters to the optimization function *lloyd_optimize_mesh_3* through these mesh generation functions.

#include <CGAL/refine_mesh_3.h>

parameters::internal::Lloyd_options

parameters::lloyd(double parameters::time_limit = 0,
std::size_t parameters::max_iteration_number = 0,
double parameters::convergence = 0.02,
double parameters::freeze_bound = 0.01)

Parameters

The parameters are named parameters. They are the same (i.e. they have the same name and the same default values) as the parameters of the *lloyd_optimize_mesh_3* function. See its manual page for further details.

Example

```
// Mesh generation with lloyd optimization step
C3t3 c3t3 = make_mesh_3<c3t3>(domain, criteria, parameters::lloyd());
refine_mesh_3(c3t3, domain, criteria,
   parameters::lloyd(parameters::time_limit=10));
```

See Also

CGAL::no_lloyd . . . . page **??**
CGAL::lloyd_optimize_mesh_3 . . . . page 3227
CGAL::make_mesh_3 . . . . page 3217
CGAL::refine_mesh_3 . . . . page 3220

Function

CGAL::parameters::no_lloyd

Definition

The function *parameters::no_lloyd* allows the user to tell the mesh generation functions *make_mesh_3* and *refine_mesh_3* that no lloyd optimization must be done.

#include <CGAL/refine_mesh_3.h>

parameters::internal::Lloyd_options

parameters::no_lloyd()

Example

```
// Mesh generation without lloyd optimization
C3t3 c3t3 = make_mesh_3<c3t3>(domain, criteria, parameters::no_lloyd());
```

See Also

CGAL::parameters::lloyd .... page 3237
CGAL::lloyd_optimize_mesh_3 .... page 3227
CGAL::make_mesh_3 .... page 3217
CGAL::refine_mesh_3 .... page 3220

CGAL::parameters::odt

Function

Definition

The function *parameters::odt* allows the user to trigger a call to *odt_optimize_mesh_3* in *make_mesh_3* and *refine_mesh_3* mesh optimization functions. It also allows the user to pass parameters to the optimization function *odt_optimize_mesh_3* through these mesh generation functions.

#include <CGAL/refine_mesh_3.h>

parameters::internal::Odt_options

parameters::odt(double parameters::time_limit = 0,
std::size_t parameters::max_iteration_number = 0,
double parameters::convergence = 0.02,
double parameters::freeze_bound = 0.01)

Parameters

The parameters are named parameters. They are the same (i.e. they have the same name and the same default values) as the parameters of *odt_optimize_mesh_3* function. See its manual page for further details.

Example

```
// Mesh generation with odt optimization step
C3t3 c3t3 = make_mesh_3<c3t3>(domain, criteria, parameters::odt());
refine_mesh_3(c3t3, domain, criteria,
   parameters::odt(parameters::time_limit=10));
```

See Also

CGAL::no_odt . . . . page **??**
CGAL::odt_optimize_mesh_3 . . . . page 3229
CGAL::make_mesh_3 . . . . page 3217
CGAL::refine_mesh_3 . . . . page 3220

Function

CGAL::parameters::no_odt

Definition

The function *parameters::no_odt* allows the user to tell the mesh generation functions *make_mesh_3* and *refine_mesh_3* that no odt optimization must be done.

#include <CGAL/refine_mesh_3.h>

parameters::internal::Odt_options

parameters::no_odt()

Example

```
// Mesh generation without odt optimization
C3t3 c3t3 = make_mesh_3<c3t3>(domain, criteria, parameters::no_odt());
```

See Also

CGAL::parameters::odt . . . . page 3239
CGAL::odt_optimize_mesh_3 . . . . page 3229
CGAL::make_mesh_3 . . . . page 3217
CGAL::refine_mesh_3 . . . . page 3220

CGAL::Mesh_optimization_return_code

Definition

The enum *Mesh_optimization_return_code* is the output of the global mesh optimization functions. This output corresponds to mesh optimization process termination reasons. Note that depending on what parameters have been set to the optimizer, each return value may represent a failure or a success.

#include <CGAL/Mesh_optimization_return_code.h>

enum Mesh_optimization_return_code { *BOUND_REACHED = 0,*
TIME_LIMIT_REACHED,
CANT_IMPROVE_ANYMORE,
CONVERGENCE_REACHED,
MAX_ITERATION_NUMBER_REACHED}

Values

The following values are defined:

- *BOUND_REACHED*: The given lower bound on mesh quality is reached.
- *TIME_LIMIT_REACHED*: The given time limit is reached.
- *CANT_IMPROVE_ANYMORE*: Mesh could not be improved anymore.
- *CONVERGENCE_REACHED*: The given convergence bound is reached.
- *MAX_ITERATION_NUMBER_REACHED*: The given maximum iteration number is reached.

See Also

CGAL::exude_mesh_3 . . . . page 3223
CGAL::perturb_mesh_3 . . . . page 3225
CGAL::lloyd_optimize_mesh_3 . . . . page 3227
CGAL::odt_optimize_mesh_3 . . . . page 3229

Enum

CGAL::Mesh_facet_topology

Definition

The enum *Mesh_facet_topology* is designed to tell which constraints have to be checked on each surface facet during the mesh refinement process.

#include <CGAL/Mesh_facet_topology.h>

enum Mesh_facet_topology { *FACET_VERTICES_ON_SURFACE = 1,*
FACET_VERTICES_ON_SAME_SURFACE_PATCH,
FACET_VERTICES_ON_SAME_SURFACE_PATCH_WITH_ADJACENCY_CHECK}

Values

The following values are defined:

- *FACET_VERTICES_ON_SURFACE*: Each vertex of the facet have to be on the surface, on a curve segment, or on a corner.
- *FACET_VERTICES_ON_SAME_SURFACE_PATCH*: The three vertices of a facet belonging to a surface patch *s* have to be on the same surface patch *s*, on a curve segment or on a corner.
- *FACET_VERTICES_ON_SAME_SURFACE_PATCH_WITH_ADJACENCY_CHECK*: The three vertices of a facet belonging to a surface patch *s* have to be on the same surface patch *s*, or on a curve segment incident to the surface patch *s* or on a corner incident to the surface patch *s*.

See Also

CGAL::Mesh_criteria_3<Tr> .... page 3199,
CGAL::Mesh_facet_criteria_3<Tr> .... page 3203.

Part XII

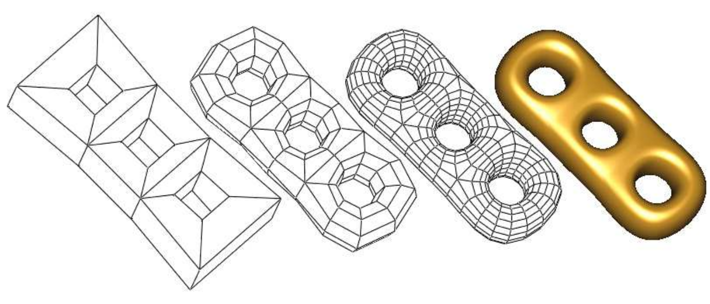
Geometry Processing

Chapter 52

3D Surface Subdivision Methods

Le-Jeng Andy Shiue

Contents



52.1 Introduction

Subdivision methods are simple yet powerful ways to generate smooth surfaces from arbitrary polyhedral meshes. Unlike spline-based surfaces (e.g NURBS) or other numeric-based modeling techniques, users of subdivision methods do not need the mathematical knowledge of the subdivision methods. The natural intuition of the geometry suffices to control the subdivision methods.

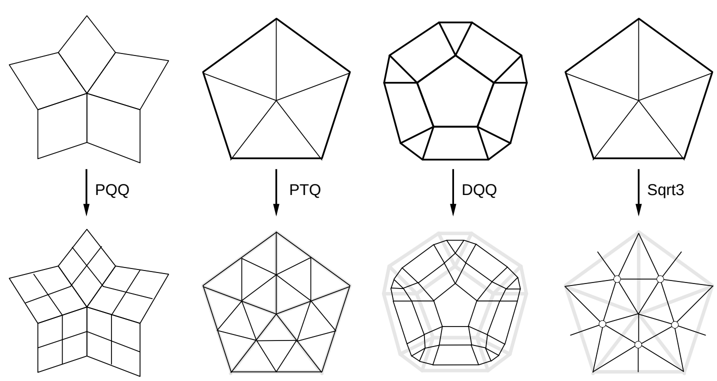
Subdivision_method_3, designed to work on the class *Polyhedron_3*, aims to be easy to use and to extend. *Subdivision_method_3* is not a class, but a namespace which contains four popular subdivision methods and their refinement functions. These include Catmull-Clark, Loop, Doo-Sabin and $\sqrt{3}$ subdivisions. Variations of these methods can be easily extended by substituting the geometry computation of the refinement host.

52.2 Subdivision Method

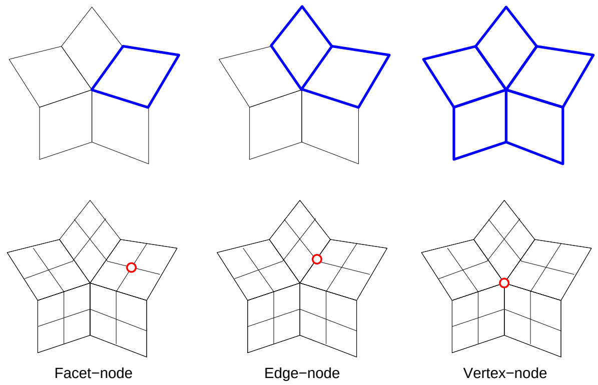
In this chapter, we explain some fundamentals of subdivision methods. We focus only on the topics that help you to understand the design of the package. [WW02] has details on subdivision methods. Some terminology introduced in this section will be used again in later sections. If you are only interested in using a specific subdivision method, Section 52.3 gives a quick tutorial on Catmull-Clark subdivision.

A subdivision method recursively refines a coarse mesh and generates an ever closer approximation to a smooth surface. The coarse mesh can have arbitrary shape, but it has to be a 2-manifold. In a 2-manifold, every interior point has a neighborhood homeomorphic to a 2D disk. Subdivision methods on non-manifolds have been developed, but are not considered in *Subdivision_method_3*. The chapter teaser shows the steps of Catmull-Clark subdivision on a CAD model. The coarse mesh is repeatedly refined by a quadrisection pattern, and new points are generated to approximate a smooth surface.

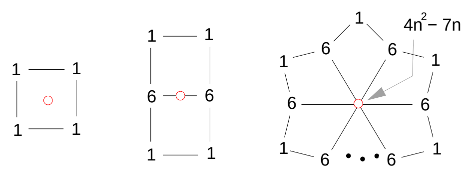
Many refinement patterns are used in practice. *Subdivision_method_3* supports the four most popular patterns, and each of them is used by Catmull-Clark[CC78], Loop, Doo-Sabin and $\sqrt{3}$ subdivision (left to right in the figure). We name these patterns by their topological characteristics instead of the associated subdivision methods. PQQ indicates the *P*rimal *Q*uadtrateral *Q*uadrisection. PTQ indicates the *P*rimal *T*riangle *Q*uadrisection. DQQ indicates the *D*ual *Q*uadtrateral *Q*uadrisection. $\sqrt{3}$ indicates the converging speed of the triangulation toward the subdivision surface.



The figure demonstrates these four refinement patterns on the 1-disk of a valence-5 vertex/facet. Refined meshes are shown below the source meshes. Points on the refined mesh are generated by averaging neighbor points on the source mesh. A graph, called *stencil*, determines the source neighborhood whose points contribute to the position of a refined point. A refinement pattern usually defines more than one stencil. For example, the PQQ refinement has a vertex-node stencil, which defines the 1-ring of an input vertex; an edge-node stencil, which defines the 1-ring of an input edge; and a facet-node stencil, which defines an input facet. The stencils of the PQQ refinement are shown in the following figure. The blue neighborhoods in the top row indicate the corresponding stencils of the refined nodes in red.



Stencils with weights are called *geometry masks*. A subdivision method defines a geometry mask for each stencil, and generates new points by averaging source points weighted by the mask. Geometry masks are carefully chosen to meet requirements of certain surface smoothness and shape quality. The geometry masks of Catmull-Clark subdivision are shown below.



The weights shown here are unnormalized, and n is the valence of the vertex. The generated point, in red, is computed by a summation of the weighted points. For example, a Catmull-Clark facet-node is computed by the summation of $1/4$ of each point on its stencil.

A stencil can have an unlimited number of geometry masks. For example, a facet-node of PQQ refinement may be computed by the summation of $1/5$ of each stencil node instead of $1/4$. Although it is legal in *Subdivision_method_3* to have any kind of geometry mask, the result surfaces may be odd, not smooth, or not even exist. [WW02] explains the details on designing masks for a quality subdivision surface.

52.3 A Quick Example: Catmull-Clark Subdivision

Assuming you are familiar with *Polyhedron_3*, you can integrate *Subdivision_method_3* into your program without much effort.

```
#include <CGAL/Simple_cartesian.h>
#include <CGAL/Subdivision_method_3.h>
#include <iostream>
#include <CGAL/Polyhedron_3.h>
#include <CGAL/IO/Polyhedron_iostream.h>

typedef CGAL::Simple_cartesian<double>     Kernel;
typedef CGAL::Polyhedron_3<Kernel>         Polyhedron;

using namespace std;
using namespace CGAL;
```

```
int main(int argc, char** argv) {
  if (argc != 2) {
    cout << "Usage: CatmullClark_subdivision d < filename" << endl;
    cout << "       d: the depth of the subdivision (0 < d < 10)" <<
   endl;
    cout << "       filename: the input mesh (.off)" << endl;
    return 0;
  }

  int d = argv[1][0] - '0';

  Polyhedron P;
  cin >> P; // read the .off

  Subdivision_method_3::CatmullClark_subdivision(P,d);

  cout << P; // write the .off

  return 0;
}
```

File: examples/Subdivision_method_3/CatmullClark_subdivision.cpp

This example demonstrates the use of the Catmull-Clark subdivision method on a *Polyhedron_3*. The polyhedron is restricted in the Cartesian space, where most subdivision applications are designed to work. There is only one line deserving a detailed explanation:

```
Subdivision_method_3::CatmullClark_subdivision(P,d);
```

Subdivision_method_3 specifies the namespace of our subdivision functions. *CatmullClark_subdivision(P,d)* computes the Catmull-Clark subdivision surface of the polyhedron *P* after *d* iterations of the refinements. The polyhedron *P* is passed by reference, and is modified (i.e. subdivided) by the subdivision function.

This example shows how to subdivide a simple *Polyhedron_3* with *Subdivision_method_3*. An application-defined polyhedron might use a specialized kernel and/or a specialized internal container. There are two major restrictions on the application-defined polyhedron to work with *Subdivision_method_3*.

- *Point_3* is type-defined by the kernel. Without *Point_3* and the associated operations being defined, *Subdivision_method_3* can not know how to compute and store the new vertex points.
- The primitives (such as vertices, halfedges and facets) in the internal container are sequentially ordered (e.g. *std::vector* and *std::list*). This implies that the iterators traverse the primitives in the order of their creations/insertions.

Section 52.5 gives detailed explanations on those two restrictions.

52.4 Catmull-Clark Subdivision

Subdivision_method_3 is designed to allow customization of the subdivision methods. This section explains the implementation of the Catmull-Clark subdivision function in *Subdivision_method_3*. The implementation demonstrates the customization of the PQQ refinement to Catmull-Clark subdivision.

When a subdivision method is developed, a refinement pattern is chosen, and then a set of geometry masks are developed to position the new points. There are three key components to implement a subdivision method:

- a mesh data structure that can represent arbitrary 2-manifolds,
- a process that refines the mesh data structure,
- and the geometry masks that compute the new points.

E. Catmull and J. Clark picked the PQQ refinement for their subdivision method, and developed a set of geometry masks to generate (or more precisely, to approximate) the B-spline surface from the control mesh. *Subdivision_method_3* provides a function that glues all three components of the Catmull-Clark subdivision method.

```
template <class Polyhedron_3, template <typename> class Mask>
void PQQ(Polyhedron_3& p, Mask<Polyhedron_3> mask, int depth)
```

Polyhedron_3 is a generic mesh data structure for arbitrary 2-manifolds. *PQQ()*, which refines the control mesh *p*, is a *refinement host* that uses a policy class *Mask<Polyhedron_3>* as part of it geometry computation. During the refinement, *PQQ()* computes and assigns new points by cooperating with the *mask*. To implement Catmull-Clark subdivision, *Mask*, the *geometry policy*, has to realize the geometry masks of Catmull-Clark subdivision. *depth* specifies the iterations of the refinement on the control mesh.

To implement the geometry masks, we need to know how a refinement host communicates with its geometry masks. The PQQ refinement defines three stencils, and hence three geometry masks are required for Catmull-Clark subdivision. The following class defines the interfaces of the stencils for the PQQ refinement.

```
template <class Polyhedron_3>
class PQQ_stencil_3 {
  void facet_node(Facet_handle facet, Point_3& pt);
  void edge_node(Halfedge_handle edge, Point_3& pt);
  void vertex_node(Vertex_handle vertex, Point_3& pt);
};
```

Each class function in *PQQ_stencil_3* computes a new point based on the neighborhood of the primitive handle, and assigns the new point to *Point_3& pt*.

We realize each class function with the geometry masks of Catmull-Clark subdivision.

```
template <class Polyhedron_3>
class CatmullClark_mask_3 {
  void facet_node(Facet_handle facet, Point_3& pt) {
    Halfedge_around_facet_circulator hcir = facet->facet_begin();
    int n = 0;
    Point_3 p(0,0,0);
    do {
      p = p + (hcir->vertex()->point() - ORIGIN);
      ++n;
    } while (++hcir != facet->facet_begin());
    pt = ORIGIN + (p - ORIGIN)/FT(n);
  }
  void edge_node(Halfedge_handle edge, Point_3& pt) {
    Point_3 p1 = edge->vertex()->point();
    Point_3 p2 = edge->opposite()->vertex()->point();
```

```
    Point_3 f1, f2;
    facet_node(edge->facet(), f1);
    facet_node(edge->opposite()->facet(), f2);
    pt = Point_3((p1[0]+p2[0]+f1[0]+f2[0])/4,
                 (p1[1]+p2[1]+f1[1]+f2[1])/4,
                 (p1[2]+p2[2]+f1[2]+f2[2])/4 );
  }
  void vertex_node(Vertex_handle vertex, Point_3& pt) {
    Halfedge_around_vertex_circulator vcir = vertex->vertex_begin();
    int n = circulator_size(vcir);

    FT Q[] = {0.0, 0.0, 0.0}, R[] = {0.0, 0.0, 0.0};
    Point_3& S = vertex->point();

    Point_3 q;
    for (int i = 0; i < n; i++, ++vcir) {
      Point_3& p2 = vcir->opposite()->vertex()->point();
      R[0] += (S[0]+p2[0])/2;
      R[1] += (S[1]+p2[1])/2;
      R[2] += (S[2]+p2[2])/2;
      facet_node(vcir->facet(), q);
      Q[0] += q[0];
      Q[1] += q[1];
      Q[2] += q[2];
    }
    R[0] /= n;    R[1] /= n;    R[2] /= n;
    Q[0] /= n;    Q[1] /= n;    Q[2] /= n;

    pt = Point_3((Q[0] + 2*R[0] + S[0]*(n-3))/n,
                 (Q[1] + 2*R[1] + S[1]*(n-3))/n,
                 (Q[2] + 2*R[2] + S[2]*(n-3))/n );
  }
};
```

This example shows the default implementation of Catmull-Clark masks in *Subdivision_method_3*. This default implementation assumes the *types* (such as *Point_3* and *Facet_handle*) are defined within *Polyhedron_3*. *CatmullClark_mask_3* is designed to work on a *Polyhedron_3* with the *Cartesian* kernel. You may need to rewrite the geometry computation to match the kernel geometry of your application.

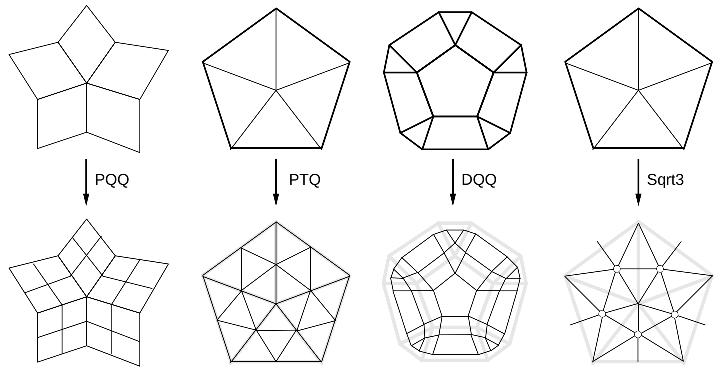
To invoke the Catmull-Clark subdivision method, we call *PQQ()* with the Catmull-Clark masks we just defined.

```
PQQ(p, CatmullClark_mask_3<Polyhedron_3>(), depth);
```

Loop, Doo-Sabin and $\sqrt{3}$ subdivisions are implemented in the similar process: pick a refinement host and implement the geometry policy. The key of developing your own subdivision method is implementing the right combination of the refinement host and the geometry policy. It is explained in the next two sections.

52.5 Refinement Host

A refinement host is a template function of a polyhedron class and a geometry mask class. It refines the input polyhedron, and computes new points through the geometry masks. *Subdivision_method_3* supports four refinement hosts: primal quadrilateral quadrisection (PQQ), primal triangle quadrisection (PTQ), dual quadrilateral quadrisection (DQQ) and $\sqrt{3}$ triangulation. Respectively, they are used by Catmull-Clark, Loop, Doo-Sabin and $\sqrt{3}$ subdivision.




```
namespace Subdivision_method_3 {
  template <class Polyhedron_3, template <typename> class Mask>
  void PQQ(Polyhedron_3& p, Mask<Polyhedron_3> mask, int step);

  template <class Polyhedron_3, template <typename> class Mask>
  void PTQ(Polyhedron_3& p, Mask<Polyhedron_3> mask, int step);

  template <class Polyhedron_3, template <typename> class Mask>
  void DQQ(Polyhedron_3& p, Mask<Polyhedron_3> mask, int step)

  template <class Polyhedron_3, template <typename> class Mask>
  void Sqrt3(Polyhedron_3& p, Mask<Polyhedron_3> mask, int step)
}
```

The polyhedron class is a specialization of *Polyhedron_3*, and the mask is a policy class realizing the geometry masks of the subdivision method.

A refinement host refines the input polyhedron, maintains the stencils (i.e., the mapping between the control mesh and the refined mesh), and calls the geometry masks to compute the new points. In *Subdivision_method_3*, refinements are implemented as a sequence of connectivity operations (mainly Euler operations). The order of the connectivity operations plays a key role when maintaining stencils. By matching the order of the source submeshes to the refined vertices, no flag in the primitives is required to register the stencils. It avoids the data dependency of the refinement host on the polyhedron class. To make the ordering trick work, the polyhedron class must have a sequential container, such as a vector or a linked-list, as the internal storage. A sequential container guarantees that the iterators of the polyhedron always traverse the primitives in the order of their insertions. Non-sequential structures such as trees or maps do not provide the required ordering, and hence can not be used with *Subdivision_method_3*.

Although *Subdivision_method_3* does not require flags to support the refinements and the stencils, it still needs to know how to compute and store the geometry data (i.e. the points). *Subdivision_method_3* expects that the typename *Point_3* is defined in the geometry kernel of the polyhedron (i.e. the *Polyhedron_3::Traits::Kernel*). A point of the type *Point_3* is returned by the geometry policy and is then assigned to the new vertex. The geometry policy is explained in next section.

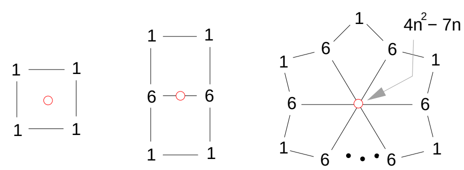
Refinement hosts *PQQ* and *DQQ* work on a general polyhedron, and *PTQ* and *Sqrt3* work on a triangulated polyhedron. The result of *PTQ* and *Sqrt3* on a non-triangulated polyhedron is undefined. *Subdivision_method_3* does not verify the precondition of the mesh characteristics before the refinement.

For details of the refinement implementation, interested users should refer to [SP05].

52.6 Geometry Policy

A geometry policy defines a set of geometry masks. Each geometry mask is realized as a member function that computes new points of the subdivision surface.

Each geometry mask receives a primitive handle (e.g. *Halfedge_handle*) of the control mesh, and returns a *Point_3* to the subdivided vertex. The function collects the vertex neighbors of the primitive handle (i.e. nodes on the stencil), and computes the new point based on the neighbors and the mask (i.e. the stencil weights).



This figure shows the geometry masks of Catmull-Clark subdivision. The weights shown here are unnormalized, and *n* is the valence of the vertex. The new points are computed by the summation of the weighted points on their stencils. Following codes show an implementation of the geometry mask of the facet-node. The complete listing of a Catmull-Clark geometry policy is in the Section 52.4.

```
template <class Polyhedron_3>
class CatmullClark_mask_3 {
  void facet_node(Facet_handle facet, Point_3& pt) {
    Halfedge_around_facet_circulator hcir = facet->facet_begin();
    int n = 0;
    Point_3 p(0,0,0);
    do {
      p = p + (hcir->vertex()->point() - ORIGIN);
      ++n;
    } while (++hcir != facet->facet_begin());
    pt = ORIGIN + (p - ORIGIN)/FT(n);
  }
}
```

In this example, the computation is based on the assumption that the *Point_3* is the *CGAL::Point_3*. It is an assumption, but not a restriction. You are allowed to use any point class as long as it is defined as the *Point_3* in your polyhedron. You may need to modify the geometry policy to support the computation and the assignment of the specialized point. This extension is not unusual in graphics applications. For example, you might want to subdivide the texture coordinates for your subdivision surface.

The refinement host of Catmull-Clark subdivision requires three geometry masks for polyhedrons without open boundaries: a vertex-node mask, an edge-node mask, and a facet-node mask. To support polyhedrons with boundaries, a border-node mask is also required. The border-node mask for Catmull-Clark subdivision is listed below, where *ept* returns the new point splitting *edge* and *vpt* returns the new point on the vertex pointed by *edge*.

```
  void border_node(Halfedge_handle edge, Point_3& ept, Point_3& vpt) {
    Point_3& ep1 = edge->vertex()->point();
    Point_3& ep2 = edge->opposite()->vertex()->point();
    ept = Point_3((ep1[0]+ep2[0])/2, (ep1[1]+ep2[1])/2,
   (ep1[2]+ep2[2])/2);
```

```
    Halfedge_around_vertex_circulator vcir = edge->vertex_begin();
    Point_3& vp1  = vcir->opposite()->vertex()->point();
    Point_3& vp0  = vcir->vertex()->point();
    Point_3& vp_1 = (--vcir)->opposite()->vertex()->point();
    vpt = Point_3((vp_1[0] + 6*vp0[0] + vp1[0])/8,
                  (vp_1[1] + 6*vp0[1] + vp1[1])/8,
                  (vp_1[2] + 6*vp0[2] + vp1[2])/8 );
  }
```

The mask interfaces of all four refinement hosts are listed below. *DQQ_stencil_3* and *Sqrt3_stencil_3* do not have the border-node stencil because the refinement hosts of DQQ and $\sqrt{3}$ refinements do not support global boundaries in the current release. This might be changed in the future releases.

```
template <class Polyhedron_3>
class PQQ_stencil_3 {
  void facet_node(Facet_handle, Point_3&);
  void edge_node(Halfedge_handle, Point_3&);
  void vertex_node(Vertex_handle, Point_3&);

  void border_node(Halfedge_handle, Point_3&, Point_3&);
};

template <class Polyhedron_3>
class PTQ_stencil_3 {
  void edge_node(Halfedge_handle, Point_3&);
  void vertex_node(Vertex_handle, Point_3&);

  void border_node(Halfedge_handle, Point_3&, Point_&);
};

template <class Polyhedron_3>
class DQQ_stencil_3 {
public:
  void corner_node(Halfedge_handle edge, Point_3& pt);
};

template <class Polyhedron_3>
class Sqrt3_stencil_3 {
public:
  void vertex_node(Vertex_handle vertex, Point_3& pt);
};
```

The source codes of *CatmullClark_mask_3*, *Loop_mask_3*, *DooSabin_mask_3*, and *Sqrt3_mask_3* are the best sources of learning these stencil interfaces.

52.7 The Four Subdivision Methods

Subdivision_method_3 supports Catmull-Clark, Loop, Doo-Sabin and $\sqrt{3}$ subdivisions by specializing their respective refinement hosts. They are designed to work on a *Polyhedron_3*. If your application uses a polyhedron with a specialized geometry kernel, you need to specialize the refinement host with a geometry policy based on that kernel.

```
namespace Subdivision_method_3 {
  template <class Polyhedron_3>
  void CatmullClark_subdivision(Polyhedron_3& p, int step = 1) {
    PQQ(p, CatmullClark_mask_3<Polyhedron_3>(), step);
  }

  template <class Polyhedron_3>
  void Loop_subdivision(Polyhedron_3& p, int step = 1) {
    PTQ(p, Loop_mask_3<Polyhedron_3>() , step);
  }

  template <class Polyhedron_3>
  void DooSabin_subdivision(Polyhedron_3& p, int step = 1) {
    DQQ(p, DooSabin_mask_3<Polyhedron_3>(), step);
  }

  template <class Polyhedron_3>
  void Sqrt3_subdivision(Polyhedron_3& p, int step = 1) {
    Sqrt3(p, Sqrt3_mask_3<Polyhedron_3>(), step);
  }
}
```

The following example demonstrates the use of Doo-Sabin subdivision on a polyhedral mesh.

```
#include <CGAL/Simple_cartesian.h>
#include <CGAL/Subdivision_method_3.h>

#include <iostream>

#include <CGAL/Polyhedron_3.h>
#include <CGAL/IO/Polyhedron_iostream.h>

typedef CGAL::Simple_cartesian<double>     Kernel;
typedef CGAL::Polyhedron_3<Kernel>         Polyhedron;

using namespace std;
using namespace CGAL;

int main(int argc, char **argv) {
  if (argc != 2) {
    cout << "Usage: DooSabin_subdivision d < filename" << endl;
    cout << "       d: the depth of the subdivision (0 < d < 10)" <<
   endl;
    cout << "       filename: the input mesh (.off)" << endl;
    return 0;
  }

  int d = argv[1][0] - '0';

  Polyhedron P;
  cin >> P; // read the .off

  Subdivision_method_3::DooSabin_subdivision(P,d);
```

```
  cout << P; // write the .off

  return 0;
}
```

File: examples/Subdivision_method_3/DooSabin_subdivision.cpp

52.8 Other Subdivision Methods

Subdivision_method_3 supports four practical subdivision methods on a Cartesian *Polyhedron_3*. More subdivision methods can be supported through the specialization of refinement hosts with custom geometry masks. The following example develops a subdivision method generating an improved Loop subdivision surface.

```
#include <CGAL/Simple_cartesian.h>
#include <CGAL/Subdivision_method_3.h>

#include <cstdio>
#include <iostream>

#include <CGAL/Polyhedron_3.h>
#include <CGAL/IO/Polyhedron_iostream.h>

typedef CGAL::Simple_cartesian<double>     Kernel;
typedef CGAL::Polyhedron_3<Kernel>         Polyhedron;

using namespace std;
using namespace CGAL;

// ======================================================================
template <class Poly>
class WLoop_mask_3 {
  typedef Poly                                              Polyhedron;

  typedef typename Polyhedron::Vertex_iterator              Vertex_iterator;
  typedef typename Polyhedron::Halfedge_iterator            Halfedge_iterator;
  typedef typename Polyhedron::Facet_iterator               Facet_iterator;

  typedef typename Polyhedron::Halfedge_around_facet_circulator

    Halfedge_around_facet_circulator;
  typedef typename Polyhedron::Halfedge_around_vertex_circulator

    Halfedge_around_vertex_circulator;

  typedef typename Polyhedron::Traits                       Traits;
  typedef typename Traits::Kernel                           Kernel;

  typedef typename Kernel::FT                               FT;
  typedef typename Kernel::Point_3                          Point;
  typedef typename Kernel::Vector_3                         Vector;
```

```
public:
  void edge_node(Halfedge_iterator eitr, Point& pt) {
    Point& p1 = eitr->vertex()->point();
    Point& p2 = eitr->opposite()->vertex()->point();
    Point& f1 = eitr->next()->vertex()->point();
    Point& f2 = eitr->opposite()->next()->vertex()->point();

    pt = Point((3*(p1[0]+p2[0])+f1[0]+f2[0])/8,
               (3*(p1[1]+p2[1])+f1[1]+f2[1])/8,
               (3*(p1[2]+p2[2])+f1[2]+f2[2])/8 );
  }
  void vertex_node(Vertex_iterator vitr, Point& pt) {
    double R[] = {0.0, 0.0, 0.0};
    Point& S = vitr->point();

    Halfedge_around_vertex_circulator vcir = vitr->vertex_begin();
    std::size_t n = circulator_size(vcir);
    for (std::size_t i = 0; i < n; i++, ++vcir) {
      Point& p = vcir->opposite()->vertex()->point();
      R[0] += p[0];     R[1] += p[1];   R[2] += p[2];
    }
    if (n == 6) {
      pt = Point((10*S[0]+R[0])/16, (10*S[1]+R[1])/16,
   (10*S[2]+R[2])/16);
    } else if (n == 3) {
      double B = (5.0/8.0 - std::sqrt(3+2*std::cos(6.283/n))/64.0)/n;
      double A = 1-n*B;
      pt = Point((A*S[0]+B*R[0]), (A*S[1]+B*R[1]), (A*S[2]+B*R[2]));
    } else {
      double B = 3.0/8.0/n;
      double A = 1-n*B;
      pt = Point((A*S[0]+B*R[0]), (A*S[1]+B*R[1]), (A*S[2]+B*R[2]));
    }
  }
  void border_node(Halfedge_iterator eitr, Point& ept, Point& vpt) {
    Point& ep1 = eitr->vertex()->point();
    Point& ep2 = eitr->opposite()->vertex()->point();
    ept = Point((ep1[0]+ep2[0])/2, (ep1[1]+ep2[1])/2, (ep1[2]+ep2[2])/2);

    Halfedge_around_vertex_circulator vcir = eitr->vertex_begin();
    Point& vp1  = vcir->opposite()->vertex()->point();
    Point& vp0  = vcir->vertex()->point();
    Point& vp_1 = (--vcir)->opposite()->vertex()->point();
    vpt = Point((vp_1[0] + 6*vp0[0] + vp1[0])/8,
                (vp_1[1] + 6*vp0[1] + vp1[1])/8,
                (vp_1[2] + 6*vp0[2] + vp1[2])/8 );
  }
};

int main(int argc, char **argv) {
  if (argc != 2) {
    cout << "Usage: Customized_subdivision d < filename" << endl;
    cout << "       d: the depth of the subdivision (0 < d < 10)" <<
   endl;
```

```
    cout << "        filename: the input mesh (.off)" << endl;
    return 0;
  }

  int d = argv[1][0] - '0';

  Polyhedron P;
  cin >> P; // read the .off

  Subdivision_method_3::PTQ(P, WLoop_mask_3<Polyhedron>(), d);

  cout << P; // write the .off

  return 0;
}
```

File: examples/Subdivision_method_3/Customized_subdivision.cpp

The points generated by the geometry mask are semantically required to converge to a smooth surface. This is the requirement imposed by the theory of the subdivision surface. *Subdivision_method_3* does not enforce this requirement, nor will it verify the smoothness of the subdivided mesh. *Subdivision_method_3* guarantees the topological properties of the subdivided mesh. A genus-n 2-manifold is assured to be subdivided into a genus-n 2-manifold. But when specialized with ill-designed geometry masks, *Subdivision_method_3* may generate a surface that is odd, not smooth, or not even exist.

3D Surface Subdivision Methods Reference Manual

Le-Jeng Andy Shiue

Subdivision methods recursively refine the control mesh (i.e. the input mesh) and generate points approximating the limit surface. Designed to work on the class *Polyhedron_3*, *Subdivision_method_3* aims to be easy to use and to extend. *Subdivision_method_3* is not a class, but a namespace which consists of four popular subdivision methods and their refinement hosts. Supported subdivision methods include Catmull-Clark, Loop, Doo-Sabin and $\sqrt{3}$ subdivisions. Their respective refinement hosts are PQQ, PTQ, DQQ and $\sqrt{3}$ refinements. Variations of those methods can be easily extended by substituting the geometry computation of the refinement host.

52.9 Classified Reference Pages

Concepts

Classes

52.10 Alphabetical List of Reference Pages

CGAL::Subdivision_method_3

Definition

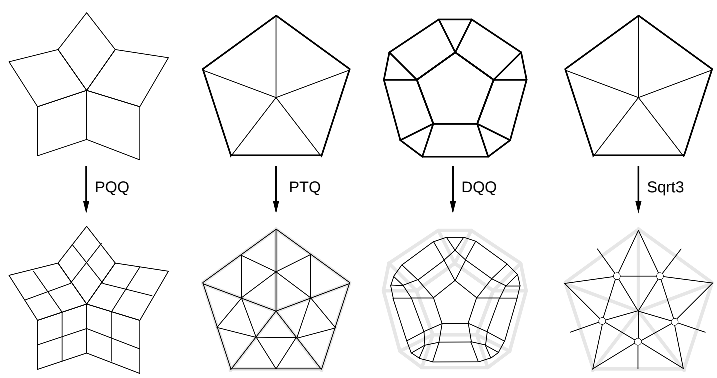
A subdivision method recursively refines a coarse mesh and generates an ever closer approximation to a smooth surface. *Subdivision_method_3* consists of four subdivision methods and their refinement hosts. Each refinement host is a template function of a polyhedron class and a geometry policy class. It refines the connectivity of the control mesh and computes the geometry of the refined mesh. The geometry computation is dedicated to the custom geometry policy. A geometry policy consists of functions that compute the new point based on the subdivision stencil. A stencil defines the footprint (a submesh of the control mesh) of a new point.

The four supported refinement hosts are the primal quadrilateral quadrisection (PQQ), the primal triangle quadrisection (PTQ), the dual quadrilateral quadrisection (DQQ), and the $\sqrt{3}$ triangulation. These refinements are respectively used in Catmull-Clark, Loop, Doo-Sabin and $\sqrt{3}$ subdivision.

#include <CGAL/Subdivision_method_3.h>

Refinement Host

A refinement host is a template function of a polyhedron class and a geometry mask class. It refines the input polyhedron, and computes new points through the geometry masks. *Subdivision_method_3* supports four refinement hosts: *PQQ*, *PTQ*, *DQQ* and *Sqrt3*.



template <class Polyhedron_3, template <typename> class Mask>
void PQQ(Polyhedron_3& p, Mask<Polyhedron_3> mask, int step = 1)

applies the PQQ refinement on the control mesh *p step* times. The geometry of the refined mesh is computed by the geometry policy *mask*. This function overwrites the control mesh *p* with the refined mesh.

template <class Polyhedron_3, template <typename> class Mask>
void PTQ(Polyhedron_3& p, Mask<Polyhedron_3> mask, int step = 1)

applies the PTQ refinement on the control mesh *p step* times, where *p* contains only triangle facets. The geometry of the refined mesh is computed by the geometry policy *mask*. This function overwrites the control mesh *p* with the refined mesh. The result of a non-triangle mesh *p* is undefined.

template <class Polyhedron_3, template <typename> class Mask>
void *DQQ(Polyhedron_3& p, Mask<Polyhedron_3> mask, int step = 1)*

applies the DQQ refinement on the control mesh *p* *step* times. The geometry of the refined mesh is computed by the geometry policy *mask*. This function overwrites the control mesh *p* with the refined mesh.

template <class Polyhedron_3, template <typename> class Mask>
void *Sqrt3(Polyhedron_3& p, Mask<Polyhedron_3> mask, int step = 1)*

applies the $\sqrt{3}$ triangulation on the control mesh *p* *step* times, where *p* contains only triangle facets. The geometry of the refined mesh is computed by the geometry policy *mask*. This function overwrites the control mesh *p* with the refined mesh. The result of a non-triangle mesh *p* is undefined.

Subdivision Method

template <class Polyhedron_3>
void *CatmullClark_subdivision(Polyhedron_3& p, int step = 1)*

applies Catmull-Clark subdivision *step* times on the control mesh *p*. This function overwrites the control mesh *p* with the subdivided mesh.

template <class Polyhedron_3>
void *Loop_subdivision(Polyhedron_3& p, int step = 1)*

applies Loop subdivision *step* times on the control mesh *p*. This function overwrites the control mesh *p* with the subdivided mesh.

template <class Polyhedron_3>
void *DooSabin_subdivision(Polyhedron_3& p, int step = 1)*

applies Doo-Sabin subdivision *step* times on the control mesh *p*. This function overwrites the control mesh *p* with the subdivided mesh.

template <class Polyhedron_3>
void *Sqrt3_subdivision(Polyhedron_3& p, int step = 1)*

applies $\sqrt{3}$ subdivision *step* times on the control mesh *p*. This function overwrites the control mesh *p* with the subdivided mesh.

See Also

CGAL::CatmullClark_mask_3<Polyhedron_3> . . . . page 3268
CGAL::Loop_mask_3<Polyhedron_3> . . . . page 3270
CGAL::Sqrt3_mask_3<Polyhedron_3> . . . . page 3272

Example

This example program subdivides a polyhedral mesh with Catmull-Clark subdivision.

```
#include <CGAL/Simple_cartesian.h>
#include <CGAL/Subdivision_method_3.h>
#include <iostream>
#include <CGAL/Polyhedron_3.h>
#include <CGAL/IO/Polyhedron_iostream.h>

typedef CGAL::Simple_cartesian<double>     Kernel;
typedef CGAL::Polyhedron_3<Kernel>         Polyhedron;

using namespace std;
using namespace CGAL;

int main(int argc, char** argv) {
  if (argc != 2) {
    cout << "Usage: CatmullClark_subdivision d < filename" << endl;
    cout << "       d: the depth of the subdivision (0 < d < 10)" <<
   endl;
    cout << "       filename: the input mesh (.off)" << endl;
    return 0;
  }

  int d = argv[1][0] - '0';

  Polyhedron P;
  cin >> P; // read the .off

  Subdivision_method_3::CatmullClark_subdivision(P,d);

  cout << P; // write the .off

  return 0;
}
```

File: examples/Subdivision_method_3/CatmullClark_subdivision.cpp

PQQMask_3

Required member functions for the PQQMask_3 concept. This policy concept of geometric computations is used in *CGAL::Subdivision_method_3::PQQ<Polyhedron_3, Mask>*.

Operations

void *mask.facet_node(Facet_handle facet, Point_3& pt)*

computes the facet-point *pt* based on the neighborhood of the facet *f*.

void *mask.edge_node(Edge_handle e, Point_3& pt)*

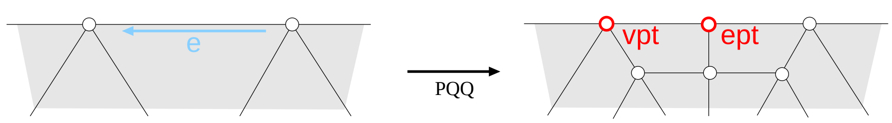
computes the edge-point *pt* based on the neighborhood of the edge *e*.

void *mask.vertex_node(Vertex_handle v, Point_3& pt)*

computes the vertex-point *pt* based on the neighborhood of the vertex *v*.

void *mask.border_node(Halfedge_handle e, Point_3& ept, Point_3& vpt)*

computes the edge-point *ept* and the vertex-point *vpt* based on the neighborhood of the border edge *e*.



Has Models

CGAL::CatmullClark_mask_3<Polyhedron_3> . . . . page 3268

See Also

CGAL::Subdivision_method_3 . . . . page 3261

PTQMask_3

Required member functions for the PTQMask_3 concept. This policy concept of geometric computations is used in *CGAL::Subdivision_method_3::PTQ<Polyhedron_3, Mask>*.

Operations

void *mask.edge_node(Edge_handle e, Point_3& pt)*

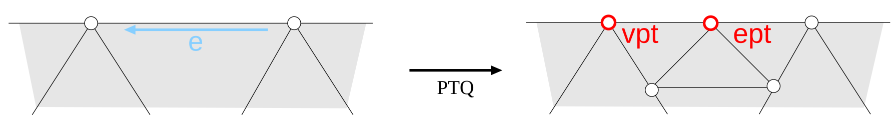
computes the edge-point *pt* based on the neighborhood of the edge *e*.

void *mask.vertex_node(Vertex_handle v, Point_3& pt)*

computes the vertex-point *pt* based on the neighborhood of the vertex *v*.

void *mask.border_node(Halfedge_handle e, Point_3& ept, Point_3& vpt)*

computes the edge-point *ept* and the vertex-point *vpt* based on the neighborhood of the border edge *e*.



Has Models

CGAL::Loop_mask_3<Polyhedron_3> . . . . page 3270

See Also

CGAL::Subdivision_method_3 . . . . page 3261

Concept

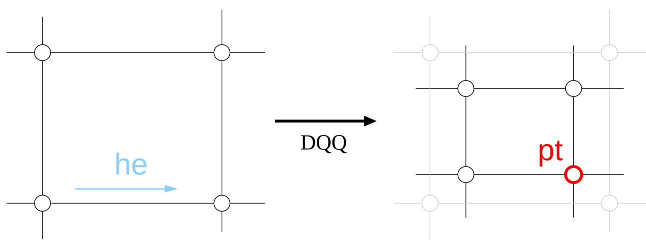
DQQMask_3

Required member functions for the DQQMask_3 concept. This policy concept of geometric computations is used in *CGAL::Subdivision_method_3::DQQ<Polyhedron_3, Mask>*.

Operations

void *mask.corner_node(Halfedge_handle he, Point_3& pt)*

computes the subdivided point *pt* based on the neighborhood of the vertex pointed by the halfedge *he*.



Has Models

CGAL::DooSabin_mask_3<Polyhedron_3> . . . . page 3271

See Also

CGAL::Subdivision_method_3 . . . . page 3261

Sqrt3Mask_3

Required member functions for the Sqrt3Mask_3 concept. This policy concept of geometric computations is used in *CGAL::Subdivision_method_3::Sqrt3<Polyhedron_3, Mask>*.

Operations

void	*mask.facet_node(Facet_handle f, Point_3& pt)*	
		computes the subdivided point *pt* based on the neighborhood of the facet *f*.
void	*mask.vertex_node(Vertex_handle v, Point& pt)*	
		computes the subdivided point *pt* based on the neighborhood of the vertex *v*.

Has Models

CGAL::Sqrt3_mask_3<Polyhedron_3> . . . . page 3272

See Also

CGAL::Subdivision_method_3 . . . . page 3261

Concept

Class

CGAL::CatmullClark_mask_3<Polyhedron_3>

Definition

A stencil determines a source neighborhood whose points contribute to the position of a refined point. The geometry mask of a stencil specifies the computation on the nodes of the stencil. *CatmullClark_mask_3<Polyhedron_3>* implements the geometry masks of Catmull-Clark subdivision on a *Polyhedron_3<Cartesian>*.

#include <CGAL/Subdivision_mask_3.h>

Parameters

The full template declaration of *CatmullClark_mask_3<Polyhedron_3>* states one template parameter:

template < class Polyhedron_3> class CatmullClark_mask_3;

The only parameter requires a *Polyhedron_3* as the argument. The *Polyhedron_3* should be specialized with the *Cartesian* kernel, which defines the *Point_3* for the vertices.

Creation

CatmullClark_mask_3<Polyhedron_3> CC; default constructor.

Stencil functions

void CC.facet_node(Facet_handle f, Point_3& pt)

computes the Catmull-Clark facet-point *pt* of the facet *f*.

void CC.edge_node(Edge_handle e, Point_3& pt)

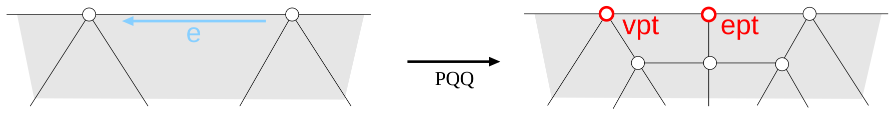
computes the Catmull-Clark edge-point *pt* of the edge *e*.

void CC.vertex_node(Vertex_handle v, Point_3& pt)

computes the Catmull-Clark vertex-point *pt* of the vertex *v*.

void CC.border_node(Halfedge_handle e, Point_3& ept, Point_3& vpt)

computes the Catmull-Clark edge-point *ept* and the Catmull-Clark vertex-point *vpt* of the border edge *e*.



See Also

CGAL::Subdivision_method_3 . . . . page 3261

Class

CGAL::Loop_mask_3<Polyhedron_3>

Definition

A stencil determines a source neighborhood whose points contribute to the position of a refined point. The geometry mask of a stencil specifies the computation on the nodes of the stencil. *Loop_mask_3<Polyhedron_3>* implements the geometry masks of Loop subdivision on a triangulated *Polyhedron_3<Cartesian>*.

#include <CGAL/Subdivision_mask_3.h>

Parameters

The full template declaration of *Loop_mask_3<Polyhedron_3>* states one template parameter:

template < class Polyhedron_3> class Loop_mask_3;

The only parameter requires a *Polyhedron_3* as the argument. The *Polyhedron_3* should be specialized with the *Cartesian* kernel, which defines the *Point_3* for the vertices.

Creation

Loop_mask_3<Polyhedron_3> L; default constructor.

Stencil functions

void *L.edge_node(Edge_handle e, Point_3& pt)*

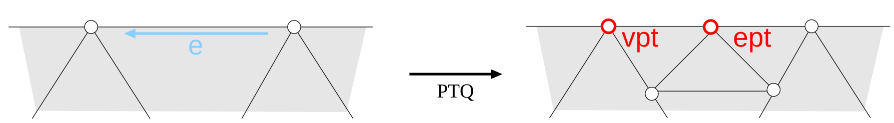
computes the Loop edge-point *pt* of the edge *e*.

void *L.vertex_node(Vertex_handle v, Point_3& pt)*

computes the Loop vertex-point *pt* of the vertex *v*.

void *L.border_node(Halfedge_handle e, Point_3& ept, Point_3& vpt)*

computes the Loop edge-point *ept* and the Loop vertex-point *vpt* of the border edge *e*.



See Also

CGAL::Subdivision_method_3 . . . . page 3261

CGAL::DooSabin_mask_3<Polyhedron_3>

Class

Definition

A stencil determines a source neighborhood whose points contribute to the position of a refined point. The geometry mask of a stencil specifies the computation on the nodes of the stencil. *DooSabin_mask_3<Polyhedron_3>* implements the geometry masks of Doo-Sabin subdivision on a *Polyhedron_3<Cartesian>*.

#include <CGAL/Subdivision_mask_3.h>

Parameters

The full template declaration of *DooSabin_mask_3<Polyhedron_3>* states one template parameter:

template < class Polyhedron_3> class DooSabin_mask_3;

The only parameter requires a *Polyhedron_3* as the argument. The *Polyhedron_3* should be specialized with the *Cartesian* kernel, which defines the *Point_3* for the vertices.

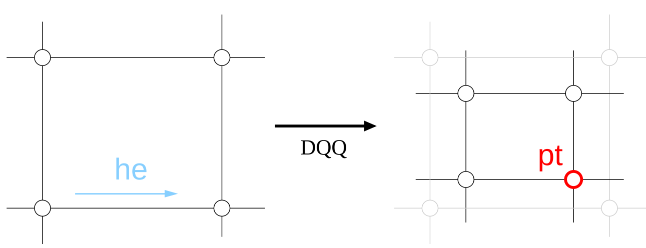
Creation

DooSabin_mask_3<Polyhedron_3> DS; default constructor.

Stencil functions

void *DS.corner_node(Halfedge_handle he, Point_3& pt)*

computes the Doo-Sabin point *pt* of the vertex pointed by the halfedge *he*.



See Also

CGAL::Subdivision_method_3 . . . . page 3261

CGAL::Sqrt3_mask_3<Polyhedron_3>

Definition

A stencil determines a source neighborhood whose points contribute to the position of a refined point. The geometry mask of a stencil specifies the computation on the nodes of the stencil. *Sqrt3_mask_3<Polyhedron_3>* implements the geometry masks of $\sqrt{3}$ subdivision on a triangulated *Polyhedron_3<Cartesian>*.

#include <CGAL/Subdivision_mask_3.h>

Parameters

The full template declaration of *Sqrt3_mask_3<Polyhedron_3>* states one template parameter:

template < class Polyhedron_3> class Sqrt3_mask_3;

The only parameter requires a *Polyhedron_3* as the argument. The *Polyhedron_3* should be specialized with the *Cartesian* kernel, which defines the *Point_3* for the vertices.

Creation

Sqrt3_mask_3<Polyhedron_3> S; default constructor.

Stencil functions

void *S.facet_node(Facet_handle f, Point_3& pt)*

computes the $\sqrt{3}$ facet-point *pt* of the facet *f*.

void *S.vertex_node(Vertex_handle v, Point& pt)*

computes the $\sqrt{3}$ vertex-point *pt* of the vertex *v*.

See Also

CGAL::Subdivision_method_3 . . . . page 3261

Chapter 53

Triangulated Surface Mesh Simplification

Fernando Cacciola

Contents

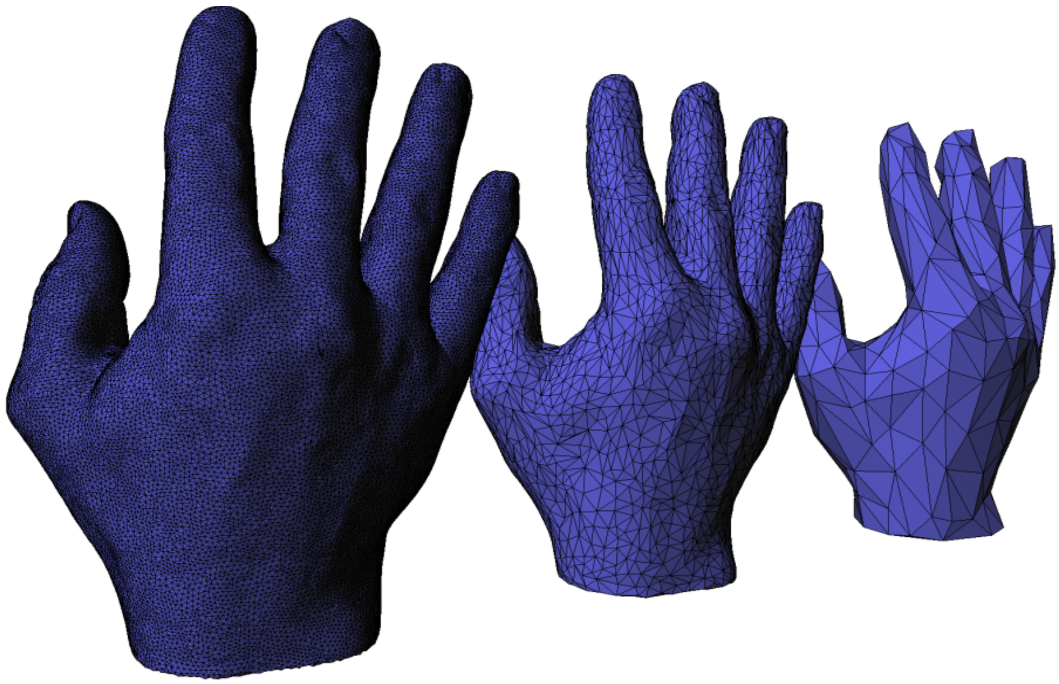
53.1 Introduction

Surface mesh simplification is the process of reducing the number of faces used in the surface while keeping the overall shape, volume and boundaries preserved as much as possible. It is the opposite of subdivision.

The algorithm presented here can simplify any *oriented 2-manifold surface*, with any number of connected components, with or without boundaries (border or holes) and handles (arbitrary genus), using a method known as *edge collapse*. Roughly speaking, the method consists of iteratively replacing an edge with a single vertex, removing 2 triangles per collapse.

Edges are collapsed according to a priority given by a user-supplied *cost* function, and the coordinates of the replacing vertex are determined by another user-supplied *placement* function. The algorithm terminates when a user-supplied *stop predicate* is met, such as reaching the desired number of edges.

The algorithm implemented here is generic in the sense that it does not require the surface to be of a particular type. Instead, it defines the concept of a *EdgeCollapsableMesh*, which presents the surface as being a halfedge data structure, and any surface that is a model of that concept can be simplified. The concept is defined not in terms of a monolithic class, but in terms of a set of functions and traits, making it easy to adapt any concrete



surface type, even if it is not a halfedge data structure at all. In particular, the concept definition follows the design of the Boost Graph Library (BGL) [SLL02].

The design is *policy-based* (http://en.wikipedia.org/wiki/Policy-based_design), meaning that you can customize some aspects of the process by passing a set of *policy objects*. Each policy object specifies a particular aspect of the algorithm, such as how edges are selected and where the replacement vertex is placed. All policies have a sensible default. Furthermore, the API uses the so-called *named-parameters* technique which allows you to pass only the relevant parameters, in any order, omitting those parameters whose default is appropriate.

53.2 Overview of the Simplification Process

The free function that implements the simplification algorithm takes not only the surface and the desired stop predicate but a number of additional parameters which control and monitor the simplification process. This section briefly describes the process in order to set the background for the discussion of the parameters to the algorithm.

There are two slightly different ”edge” collapse operations. One is known as *edge-collapse* while the other is known as *halfedge-collapse*. Given an edge ’e’ joining vertices ’w’ and ’v’, the edge-collapse operation replaces ’e’,’w’ and ’v’ for a new vertex ’r’, while the halfedge-collapse operation pulls ’v’ into ’w’, dissapearing ’e’ and leaving ’w’ in place. In both cases the operation removes the edge ’e’ along with the 2 triangles adjacent to it.

This package uses the halfedge-collapse operation, which is implemented by removing, additionally, 1 vertex (’v’) and 2 edges, one per adjacent triangle. It optionally moves the remaining vertex (’w’) into a new position, called *placement*, in which case the net effect is the same as in the edge-collapse operation.

Naturally, the surface that results from an edge collapse deviates from the initial surface by some amount, and since the goal of simplification is to reduce the number of triangles while retaining the overall look of the surface as much as possible, it is necessary to measure such a deviation. Some methods attempt to measure the total deviation from the initial surface to the completely simplified surface, for example, by tracking an accumulated error while keeping a history of the simplification changes. Other methods, like the one implemented in this package, attempt to measure only the *cost* of each individual edge collapse (the local deviation introduced by a single simplification step) and plan the entire process as a sequence of steps of increasing cost.

Global error tracking methods produce highly accurate simplifications but take up a lot of additional space. Cost-driven methods, like the one in this package, produce slightly less accurate simplifications but take up much less additional space, even none in some cases.

The cost-driven method implemented in this package is mainly based on [LT98, LT99], with contributions from [HDD^{+}93], [GH97] and [DEGN99].

The algorithm proceeds in two stages. In the first stage, called *collection stage*, an initial *collapse cost* is assigned to each and every edge in the surface. Then in the second stage, called *collapsing stage*, edges are processed in order of increasing cost. Some processed edges are collapsed while some are just discarded. Collapsed edges are replaced by a vertex and the collapse cost of all the edges now incident on the replacement vertex is recalculated, affecting the order of the remaining unprocessed edges.

Not all edges selected for processing are collapsed. A processed edge can be discarded right away, without being collapsed, if it doesn't satisfy certain topological and geometric conditions.

The algorithm presented in [GH97] contracts (collapses) arbitrary vertex pairs and not only edges by considering certain vertex pairs as forming a pseudo-edge and proceeding to collapse both edges and pseudo-edges in the same way as in [LT98, LT99] (which is the algorithm implemented here). However, contracting an arbitrary vertex-pair may result in a non-manifold surface, but the current state of this package can only deal with manifold surfaces, thus, it can only collapse edges. That is, this package cannot be used as a framework for vertex contraction.

53.3 Cost Strategy

The specific way in which the collapse cost and vertex placement is calculated is called the *cost strategy*. The user can choose different strategies in the form of policies and related parameters, passed to the algorithm.

The current version of the package provides a set of policies implementing two strategies: the Lindstrom-Turk strategy, which is the default, and a strategy consisting of an edge-length cost with an optional midpoint placement (much faster but less accurate).

53.3.1 Lindstrom-Turk Cost and Placement Strategy

The main characteristic of the strategy presented in [LT98, LT99] is that the simplified surface is not compared at each step with the original surface (or the surface at a previous step) so there is no need to keep extra information, such as the original surface or a history of the local changes. Hence the name *memoryless* simplification.

At each step, all remaining edges are potential candidates for collapsing and the one with the lowest cost is selected.

The cost of collapsing an edge is given by the position chosen for the vertex that replaces it.

The replacement vertex position is computed as the solution to a system of 3 linearly-independent linear equality constraints. Each constraint is obtained by minimizing a quadratic objective function subject to the previously computed constraints.

There are several possible candidate constraints and each is considered in order of importance. A candidate constraint might be *incompatible* with the previously accepted constraints, in which case it is rejected and the next constraint is considered.
Once 3 constraints have been accepted, the system is solved for the vertex position.

The first constraints considered preserves the shape of the surface boundaries (in case the edge profile has boundary edges). The next constraints preserve the total volume of the surface. The next constraints, if needed, optimize the local changes in volume and boundary shape. Lastly, if a constraint is still needed (because the ones previously computed were incompatible), a third (and last) constraint is added to favor equilateral triangles over elongated triangles.

The cost is then a weighted sum of the shape, volume and boundary optimization terms, where the user specifies the unit *weighting unit factor* for each term.

The local changes are computed independently for each edge using only the triangles currently adjacent to it at the time when the edge is about to be collapsed, that is, after all previous collapses. Thus, the transitive path of minimal local changes yields at the end a global change reasonably close to the absolute minimum.

53.3.2 Cost Strategy Policies

The cost strategy used by the algorithm is selected by means of two policies: *GetPlacement* and *GetCost*.

The *GetPlacement* policy is called to compute the new position for the remaining vertex after the halfedge-collapse. It returns an optional value, which can be absent, in which case the remaining vertex is kept in its place.

The *GetCost* policy is called to compute the cost of collapsing an edge. This policy uses the placement to compute the cost (which is an error measure) and determines the ordering of the edges.

The algorithm maintains an internal data structure (a mutable priority queue) which allows each edge to be processed in increasing cost order. Such a data structure requires some per-edge additional information, such as the edge's cost. If the record of per-edge additional information occupies N bytes of storage, simplifying a surface of 1 million edges (a normal size) requires 1 million times N bytes of additional storage. Thus, to minimize the amount of additional memory required to simplify a surface only the cost is attached to each edge and nothing else.

But this is a tradeoff: the cost of a collapse is a function of the placement (the new position chosen for the remaining vertex) so before *GetCost* is called for each and every edge, *GetPlacement* must also be called to obtain the placement parameter to the cost function. But that placement, which is a 3D Point, is not attached to each and every edge since that would easily *triple* the additional storage requirement.

On the one hand, this dramatically saves on memory but on the other hand is a processing waste because when an edge is effectively collapsed, *GetPlacement* must be called *again* to know were to move the remaining vertex.

Earlier prototypes shown that attaching the placement to the edge, thus avoiding one redundant call to the placement function after the edge collapsed, has little impact on the total running time. This is because the cost of an each edge is not just computed once but changes several times during the process so the placement function must be called several times just as well. Caching the placement can only avoid the very last call, when the edge is collapsed, but not all the previous calls which are needed because the placement (and cost) changes.

53.4 API

53.4.1 API Overview

Since the algorithm is free from robustness issues there is no need for exact predicates nor constructions and *Simple_cartesian*<*double*> can be used safely. [1]

The simplification algorithm is implemented as the free template function *CGAL::Surface_mesh_simplification::edge_collapse*. The function has two mandatory and several optional parameters.

Mandatory Parameters

There are two main parameters to the algorithm: the surface to be simplified (in-place) and the stop predicate.

The surface to simplify must be a model of the *EdgeCollapsableMesh* concept. Many concrete surface types, such as *CGAL::Polyhedron_3* with only triangular faces, become models of that concept via a technique known as *external adaptation*, which is described in [SLL02] and this BGL web page: `http://www.boost.org/libs/graph/doc/leda_conversion.html`

External adaptation is a way to add an interface to an object without coercing the type of the object (which happens when you adapt it by means of a wrapper). That is, the formal parameter to the *edge_collapse* function that implements the simplification is the concrete surface object itself, not an adaptor which delegates the functionality to the concrete type.

The stop predicate is called after each edge is selected for processing, *before* it is classified as collapsible or not (thus before it is collapsed). If the stop predicate returns *true* the algorithm terminates.

Optional Named Parameters

The notion of *named parameters* was also introduced in the BGL. You can read about it in [SLL02] or the following site: `http://www.boost.org/libs/graph/doc/bgl_named_params.html`. Named parameters allow the user to specify only those parameters which are really needed, by name, making the parameter ordering unimportant.

Say there is a function *f()* that takes 3 parameters called *name*, *age* and *gender*, and you have variables *n,a and g* to pass as parameters to that function. Without named parameters, you would call it like this: *f(n,a,g)*, but with named parameters, you call it like this: *f(name(n).age(a).gender(g))*.

That is, you give each parameter a name by wrapping it into a function whose name matches that of the parameter. The entire list of named parameters is really a composition of function calls separated by a dot (.). Thus, if the function takes a mix of mandatory and named parameters, you use a comma to separate the last non-named parameter from the first named parameters, like this:

f(non_named_par0, non_named_pa1, name(n).age(a).gender(g))

When you use named parameters, the ordering is irrelevant, so this: *f(name(n).age(a).gender(g))* is equivalent to this: *f(age(a).gender(g).name(n))*, and you can just omit any named parameter that has a default value.

[1] In the current version, 3.3, the LindstromTurk policies are not implemented for homogeneous coordinates, so a cartesian kernel must be used.

Sample Call

```
/*
surface                    : the surface to simplify
stop_predicate             : policy indicating when the simplification must
     finish
vertex_index_map(vimap)    : property-map giving each vertex a unique
    integer index
edge_index_map(eimap)      : property-map giving each edge a unique integer
     index
edge_is_border_map(ebmap): property-map specifying whether an edge is a
    border edge or not
get_cost(cf)               : function object computing the cost of a
    collapse
get_placement(pf)          : function object computing the placement for
    the remaining vertex
visitor(vis)               : function object tracking the simplification
    process
*/
int r = edge_collapse(surface
                     ,stop_predicate
                     ,vertex_index_map(vimap)
                       .edge_index_map(eimap)
                       .edge_is_border_map(ebmap)
                       .get_cost(cf)
                       .get_placement(pf)
                       .visitor(vis)
                     );
```

53.4.2 Examples

Example Using a Default Polyhedron

The following example illustrates the simplest of the cases. It uses an ordinary polyhedron and only one of the optional parameters. The unspecified cost strategy defaults to Lindstrom-Turk.

```
#include <iostream>
#include <fstream>

#include <CGAL/Simple_cartesian.h>
#include <CGAL/Polyhedron_3.h>
#include <CGAL/IO/Polyhedron_iostream.h>

// Adaptor for Polyhedron_3
#include <CGAL/Surface_mesh_simplification/HalfedgeGraph_Polyhedron_3.h>

// Simplification function
#include <CGAL/Surface_mesh_simplification/edge_collapse.h>

// Stop-condition policy
#include
    <CGAL/Surface_mesh_simplification/Policies/Edge_collapse/Count_stop_predicate.h>
```

```
typedef CGAL::Simple_cartesian<double> Kernel;
typedef CGAL::Polyhedron_3<Kernel> Surface;

namespace SMS = CGAL::Surface_mesh_simplification ;

int main( int argc, char** argv )
{
  Surface surface;

  std::ifstream is(argv[1]) ; is >> surface ;

  // This is a stop predicate (defines when the algorithm terminates).
  // In this example, the simplification stops when the number of
   undirected edges
  // left in the surface drops below the specified number (1000)
  SMS::Count_stop_predicate<Surface> stop(1000);

  // This the actual call to the simplification algorithm.
  // The surface and stop conditions are mandatory arguments.
  // The index maps are needed because the vertices and edges
  // of this surface lack an "id()" field.
  int r = SMS::edge_collapse
            (surface
            ,stop

   ,CGAL::vertex_index_map(boost::get(CGAL::vertex_external_index,surface))
                  .edge_index_map  (boost::get(CGAL::edge_external_index
    ,surface))
            );

  std::cout << "\nFinished...\n" << r << " edges removed.\n"
            << (surface.size_of_halfedges()/2) << " final edges.\n" ;

  std::ofstream os( argc > 2 ? argv[2] : "out.off" ) ; os << surface ;

  return 0 ;
}

// EOF //
```

File: examples/Surface_mesh_simplification/edge_collapse_polyhedron.cpp

Example Using an Enriched Polyhedron

The following example is equivalent to the previous example but using an enriched polyhedron whose halfedges support an *id* field to store the edge index needed by the algorithm. It also shows how to explicitly specify a cost strategy (other than the default) and how to use a visitor object to track the simplification process.

```
#include <iostream>
#include <fstream>
```

```
#include <CGAL/Simple_cartesian.h>
#include <CGAL/Polyhedron_3.h>
#include <CGAL/IO/Polyhedron_iostream.h>

// Adaptor for Polyhedron_3
#include <CGAL/Surface_mesh_simplification/HalfedgeGraph_Polyhedron_3.h>

// Simplification function
#include <CGAL/Surface_mesh_simplification/edge_collapse.h>

// Visitor base
#include <CGAL/Surface_mesh_simplification/Edge_collapse_visitor_base.h>

// Extended polyhedron items which include an id() field
#include <CGAL/Polyhedron_items_with_id_3.h>

// Stop-condition policy
#include
   <CGAL/Surface_mesh_simplification/Policies/Edge_collapse/Count_ratio_stop_predicate.h>

// Non-default cost and placement policies
#include
   <CGAL/Surface_mesh_simplification/Policies/Edge_collapse/Midpoint_and_length.h>

typedef CGAL::Simple_cartesian<double> Kernel;

typedef Kernel::Point_3 Point ;

//
// Setup an enriched polyhedron type which stores an id() field in the
   items
//
typedef CGAL::Polyhedron_3<Kernel,CGAL::Polyhedron_items_with_id_3>
   Surface;

typedef Surface::Halfedge_handle Halfedge_handle ;
typedef Surface::Vertex_handle   Vertex_handle ;

namespace SMS = CGAL::Surface_mesh_simplification ;

typedef SMS::Edge_profile<Surface> Profile ;


// The following is a Visitor that keeps track of the simplification
   process.
// In this example the progress is printed real-time and a few statistics
    are
// recorded (and printed in the end).
//
struct Stats
{
  Stats()
    : collected(0)
    , processed(0)
```

```
    , collapsed(0)
    , non_collapsable(0)
    , cost_uncomputable(0)
    , placement_uncomputable(0)
  {}

  std::size_t collected ;
  std::size_t processed ;
  std::size_t collapsed ;
  std::size_t non_collapsable ;
  std::size_t cost_uncomputable  ;
  std::size_t placement_uncomputable ;
} ;

struct My_visitor : SMS::Edge_collapse_visitor_base<Surface>
{
  My_visitor( Stats* s) : stats(s){}

  // Called during the collecting phase for each edge collected.
  void OnCollected( Profile const&, boost::optional<double> const& )
  {
    ++ stats->collected ;
    std::cerr << "\rEdges collected: " << stats->collected << std::flush
   ;
  }

  // Called during the processing phase for each edge selected.
  // If cost is absent the edge won't be collapsed.
  void OnSelected(Profile const&
                 ,boost::optional<double> cost
                 ,std::size_t             initial
                 ,std::size_t             current
                 )
  {
    ++ stats->processed ;
    if ( !cost )
      ++ stats->cost_uncomputable ;

    if ( current == initial )
      std::cerr << "\n" << std::flush ;
    std::cerr << "\r" << current << std::flush ;
  }

  // Called during the processing phase for each edge being collapsed.
  // If placement is absent the edge is left uncollapsed.
  void OnCollapsing(Profile const&
                   ,boost::optional<Point>  placement
                   )
  {
    if ( !placement )
      ++ stats->placement_uncomputable ;
  }

  // Called for each edge which failed the so called link-condition,
```

```
  // that is, which cannot be collapsed because doing so would
  // turn the surface into a non-manifold.
  void OnNonCollapsable( Profile const& )
  {
    ++ stats->non_collapsable;
  }

  // Called AFTER each edge has been collapsed
  void OnCollapsed( Profile const&, Vertex_handle hv )
  {
    ++ stats->collapsed;
  }

  Stats* stats ;
} ;


int main( int argc, char** argv )
{
  Surface surface;

  std::ifstream is(argv[1]) ; is >> surface ;

  // The items in this polyhedron have an "id()" field
  // which the default index maps used in the algorithm
  // need to get the index of a vertex/edge.
  // However, the Polyhedron_3 class doesn't assign any value to
  // this id(), so we must do it here:
  int index = 0 ;

  for( Surface::Halfedge_iterator eb = surface.halfedges_begin()
     , ee = surface.halfedges_end()
     ; eb != ee
     ; ++ eb
     )
    eb->id() = index++;

  index = 0 ;
  for( Surface::Vertex_iterator vb = surface.vertices_begin()
     , ve = surface.vertices_end()
     ; vb != ve
     ; ++ vb
     )
    vb->id() = index++;

  // In this example, the simplification stops when the number of
   undirected edges
  // drops below 10% of the initial count
  SMS::Count_ratio_stop_predicate<Surface> stop(0.1);

  Stats stats ;

  My_visitor vis(&stats) ;
```

```
  // The index maps are not explicitelty passed as in the previous
  // example because the surface items have a proper id() field.
  // On the other hand, we pass here explicit cost and placement
  // function which differ from the default policies, ommited in
  // the previous example.
  int r = SMS::edge_collapse
           (surface
           ,stop
           ,CGAL::get_cost      (SMS::Edge_length_cost  <Surface>())
                 .get_placement(SMS::Midpoint_placement<Surface>())
                 .visitor      (vis)
           );

  std::cout << "\nEdges collected: "  << stats.collected
            << "\nEdges proccessed: " << stats.processed
            << "\nEdges collapsed: "  << stats.collapsed
            << std::endl
            << "\nEdges not collapsed due to topological constrians: "
   << stats.non_collapsable
            << "\nEdge not collapsed due to cost computation constrians:
   "  << stats.cost_uncomputable
            << "\nEdge not collapsed due to placement computation
   constrians: " << stats.placement_uncomputable
            << std::endl ;

  std::cout << "\nFinished...\n" << r << " edges removed.\n"
            << (surface.size_of_halfedges()/2) << " final edges.\n" ;

  std::ofstream os( argc > 2 ? argv[2] : "out.off" ) ; os << surface ;

  return 0 ;
}

// EOF //
```

File: examples/Surface_mesh_simplification/edge_collapse_enriched_polyhedron.cpp

Example with edges marked as non-removable

The following example shows how to use the optional named parameter *edge_is_border_map* to prevent edges from being removed even if they are not really borders.

```
#include <iostream>
#include <fstream>

#include <CGAL/Simple_cartesian.h>
#include <CGAL/Polyhedron_3.h>
#include <CGAL/IO/Polyhedron_iostream.h>

// Adaptor for Polyhedron_3
#include <CGAL/Surface_mesh_simplification/HalfedgeGraph_Polyhedron_3.h>
```

```
// Simplification function
#include <CGAL/Surface_mesh_simplification/edge_collapse.h>

// Stop-condition policy
#include
    <CGAL/Surface_mesh_simplification/Policies/Edge_collapse/Count_stop_predicate.h>

// Map used to mark edges as fixed
#include <CGAL/Unique_hash_map.h>

typedef CGAL::Simple_cartesian<double> Kernel;
typedef CGAL::Polyhedron_3<Kernel> Surface;

namespace SMS = CGAL::Surface_mesh_simplification ;

//
// BGL property map which indicates whether an edge is border OR is
    marked as non-removable
//
class Constrains_map : public boost::put_get_helper<bool,Constrains_map>
{
public:

  typedef boost::readable_property_map_tag                    category;
  typedef bool                                                value_type;
  typedef bool                                                reference;
  typedef boost::graph_traits<Surface const>::edge_descriptor key_type;

  Constrains_map() : mConstrains(false) {}

  reference operator[](key_type const& e) const { return e->is_border()
    || is_constrained(e) ; }

  void set_is_constrained ( key_type const& e, bool is )  {
   mConstrains[e]=is; }

  bool is_constrained( key_type const& e ) const { return
   mConstrains.is_defined(e) ? mConstrains[e] : false ; }

private:

  CGAL::Unique_hash_map<key_type,bool> mConstrains ;

};

int main( int argc, char** argv )
{
  Surface surface;

  std::ifstream is(argv[1]) ; is >> surface ;

  // This is a stop predicate (defines when the algorithm terminates).
  // In this example, the simplification stops when the number of
    undirected edges
```

```
  // left in the surface drops below the specified number (1000)
  SMS::Count_stop_predicate<Surface> stop(10);

  Constrains_map constrains_map ;


  // This example marks ALL edges as non-removable, but a real world
    application would mark only selected ones.
  for( Surface::Halfedge_iterator eb = surface.halfedges_begin(), ee =
    surface.halfedges_end() ; eb != ee ; ++ eb )
     constrains_map.set_is_constrained(eb,true);

  // This the actual call to the simplification algorithm.
  // The surface and stop conditions are mandatory arguments.
  // The index maps are needed because the vertices and edges
  // of this surface lack an "id()" field.
  int r = SMS::edge_collapse
            (surface
            ,stop

    ,CGAL::vertex_index_map(boost::get(CGAL::vertex_external_index,surface))
                  .edge_index_map  (boost::get(CGAL::edge_external_index
     ,surface))
                  .edge_is_border_map(constrains_map)
            );

  std::cout << "\nFinished...\n" << r << " edges removed.\n"
            << (surface.size_of_halfedges()/2) << " final edges.\n" ;

  std::ofstream os( argc > 2 ? argv[2] : "out.off" ) ; os << surface ;

  return 0 ;
}
```

File: examples/Surface_mesh_simplification/edge_collapse_constrained_polyhedron.cpp

Triangulated Surface Mesh Simplification

Reference Manual

This chapter introduces the concepts and classes of the triangulated surface mesh simplification package. The packages provides a policy-based algorithm for the simplification of triangulated surface meshes by iterative edge-collapsing.

53.5 Classified Reference Pages

Concepts

Functions

Classes

53.6 Alphabetical List of Reference Pages

EdgeCollapsableMesh

Concept

Definition

The concept EdgeCollapsableMesh describes the requirements for the type of triangulated surface mesh that can be passed to the simplification algorithm.

The surface must be structurally equivalent to a polyhedral surface having only triangular faces. It can have any number of connected components, boundaries (borders and holes) and handles (arbitrary genus).

Refines

HalfedgeGraph

Valid Expressions

The mesh simplification algorithm requires the free function *collapse_triangulation_edge*.

template<class EdgeCollapsableMesh>
typename boost::graph_traits<EdgeCollapsableMesh>::vertex_descriptor

halfedge_collapse(typename boost::graph_traits<EdgeCollapsableMesh>::edge_descriptor const& ue, EdgeCollapsableMesh& mesh)

Collapses the undirected edge *(v0v1,v1v0)* replacing it with *v0* or *v1*, as described in the following paragraph.
Precondition: This function requires *mesh* to be an oriented 2-manifold with or without boundaries. Furthermore, the undirected edge *(v0v1,v1v0)* must satisfy the *link condition* [DEGN99], which guarantees that the surface is also 2-manifold after the edge collapse.

Let *v0* be the source and *v1* be the target vertices of *v0v1*.

For $e \in \{$ *v0v1,v1v0* $\}$, let *en* and *ep* be the next and previous edges, that is *en* = *next_edge(e, mesh)*, *ep* = *prev_edge(e,mesh)*, and let *eno* and *epo* be their opposite edges, that is *eno* = *opposite_edge(en, mesh)* and *epo* = *opposite_edge(ep,mesh)*.

Then, after the collapse of *(v0v1,v1v0)* the following holds:

- The edge *e* is no longer in *mesh*.
- One of $\{$*v0,v1*$\}$ is no longer in *mesh* while the other remains. [2] Let *vgone* be the removed vertex and *vkept* be the remaining vertex.
- If *e* was a border edge, that is *get(is_border, e, mesh) == true*, then *next_edge(ep) == en*, and *prev_edge(en) == ep*.
- If *e* was not a border edge, that is *get(is_border, e, mesh) == false*, then *ep* and *epo* are no longer in *mesh* while *en* and *eno* are kept in *mesh*.

[2] Even though it would appear that v0 can always be the vertex being removed, there is a case when removing the edge *e requires v1* to be removed as well. See figure 53.5.

- For all edges *ie* in *in_edges(vgone,mesh)*, *target(ie,mesh) == vkept* and *source(opposite_edge(ie),mesh) == vkept*.
- No other incidence information has changed in *mesh*.

The function returns vertex *vkept* (which can be either *v0* or *v1*).

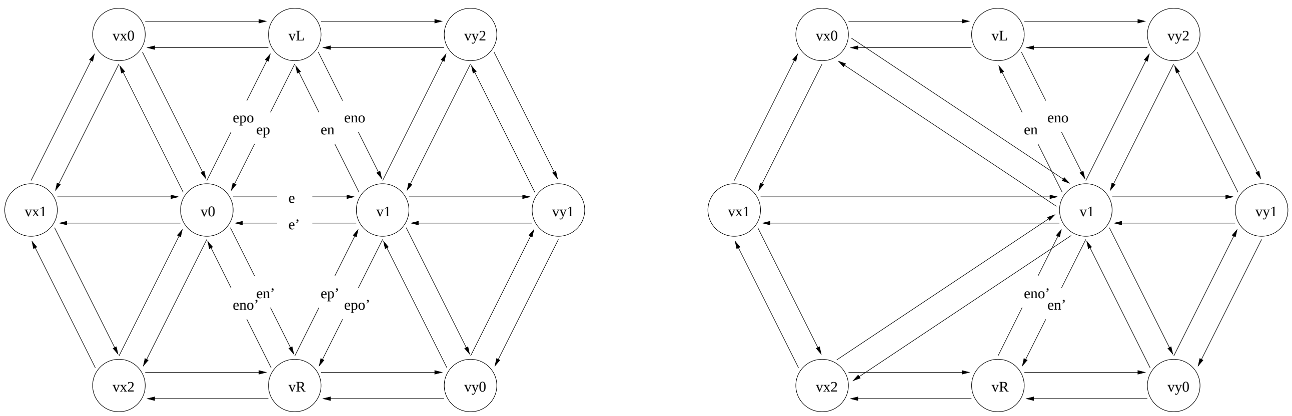


Figure 53.1: General case. The following mesh elements are removed: triangles $(v0, v1, vL)$ and $(v1, v0, vR)$, edges (e, e'), (ep, epo) and (ep', epo'), and vertex $v0$.

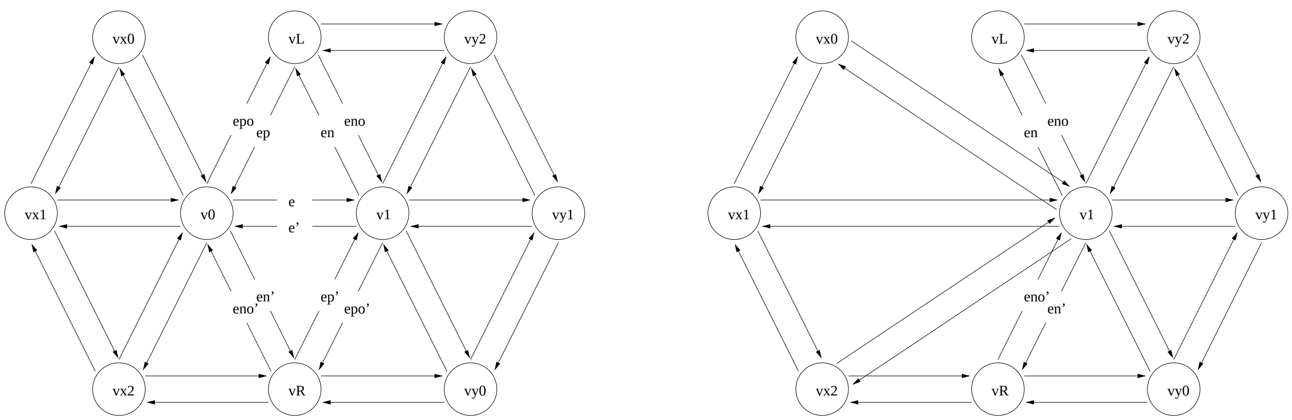


Figure 53.2: When the collapsing edge is not itself a border, but is incident upon a border edge that is removed, the operation is the same as in the general case.

Has Models

CGAL::Polyhedron_3<Traits> . . . . page 1795
(If it has only triangular faces, and via *External Adaptation*, which is described in [SLL02] and this BGL web page: `http://www.boost.org/libs/graph/doc/leda_conversion.html`).

See Also

boost::graph_traits< CGAL::Polyhedron_3<Traits> > . . . . page 4282
CGAL::halfedge_graph_traits< CGAL::Polyhedron_3<Traits> > . . . . page 4286

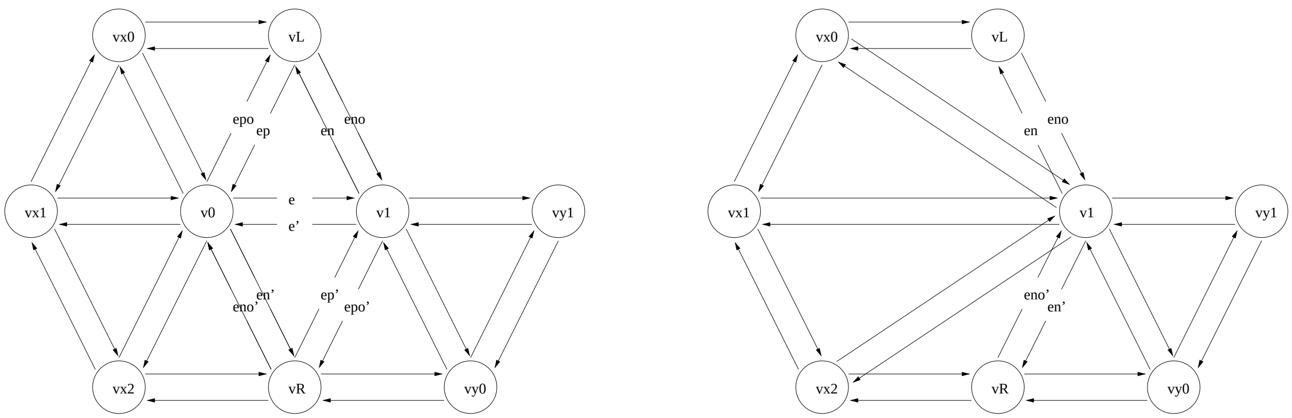


Figure 53.3: When the collapsing edge is not itself a border, but is incident upon a border edge that is *not* removed, the operation is still the same as in the general case.

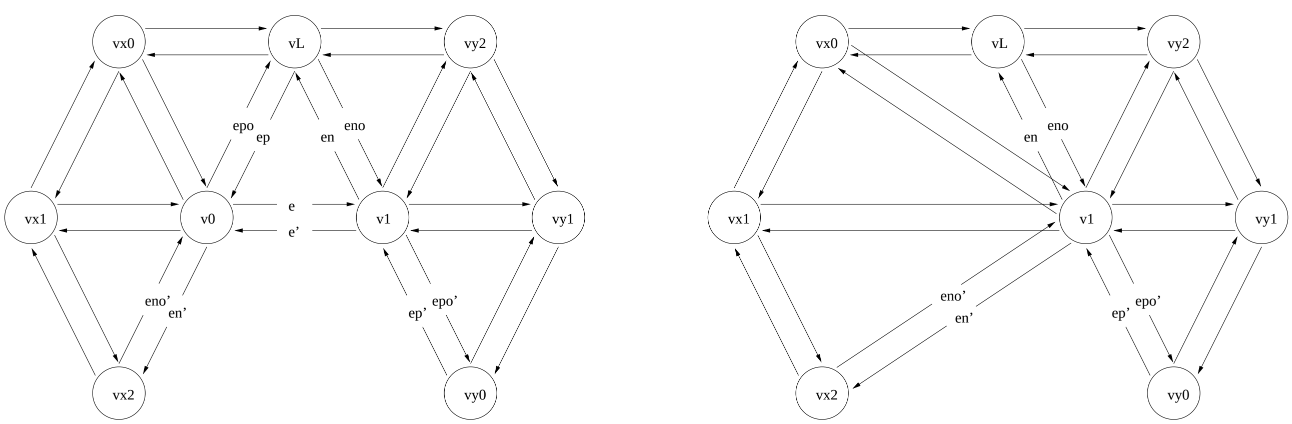


Figure 53.4: When the collapsing edge is itself a border, only 1 triangle is removed. Thus, even if (ep', epo') exists, it's not removed.

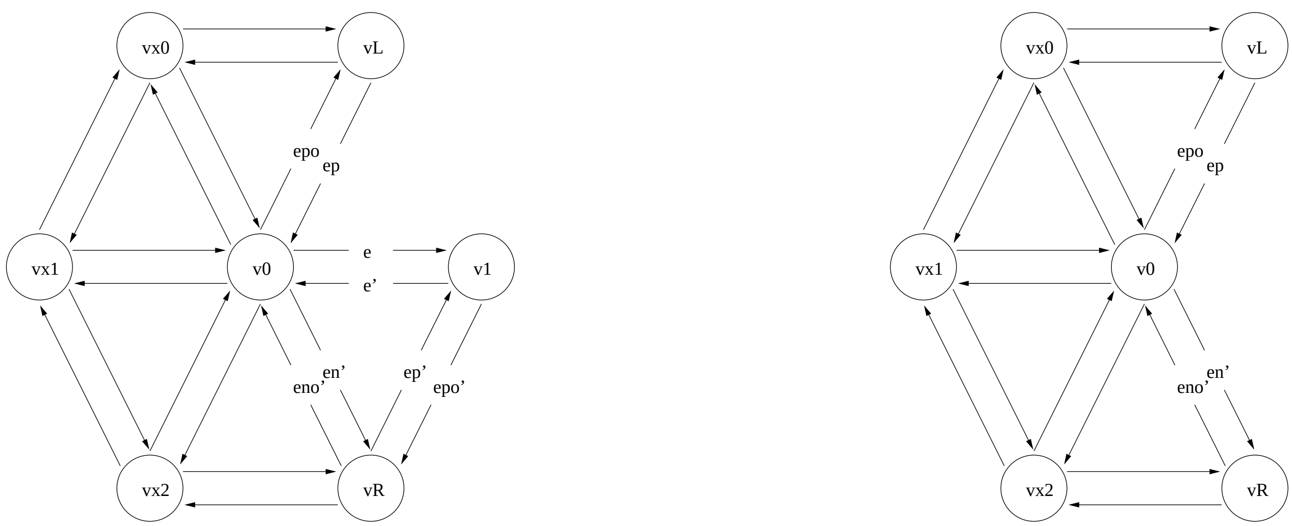


Figure 53.5: This figure illustrates the single exceptional case when removing $(v0, v1)$ neccesarily implies removing $(v1)$, thus $(v0)$ remains.

EdgeProfile

Concept

Definition

The concept EdgeProfile describes the requirements for a data structure that caches the local topology and geometry in the surroundings of an undirected edge.

This profile is used by the stop, cost and placement policies.

Types

EdgeProfile:: ECM		The type of the surface to simplify. Must be a model of the *EdgeCollapsableMesh* concept.
typename boost::graph_traits<ECM>::vertex_descriptor		
	vertex_descriptor;	A BGL vertex descriptor representing a vertex of the surface.
typename boost::graph_traits<ECM>::edge_descriptor		
	edge_descriptor;	A BGL edge descriptor representing an edge of the surface.
typename CGAL::halfedge_graph_traits<ECM>::Point		
	Point;	The point type for the surface vertex. Must be a model of *Point_3*.

Access Functions

vertex_descriptor	*ep.v0() const*	One of vertices of the edge to be collapsed.
vertex_descriptor	*ep.v1() const*	The other vertex of the edge to be collapsed.
edge_descriptor	*ep.v0_v1() const*	One of the directed edges corresponding to the undirected edge being collapsed.
edge_descriptor	*ep.v1_v0() const*	The other directed edge corresponding to the undirected edge being collapsed.
Point	*ep.p0() const*	The point of vertex $v0$.
Point	*ep.p1() const*	The point of vertex $v1$.
vertex_descriptor	*ep.vL() const*	If $v0v1$ belongs to a finite face (is not a border edge) the third vertex of that triangular face, a *null descriptor* otherwise.
edge_descriptor	*ep.v_1vL() const*	If $v0v1$ belongs to a finite face (is not a border edge) the directed edge from $v1$ to vL, a *null descriptor* otherwise.
edge_descriptor	*ep.vL_v0() const*	If $v0v1$ belongs to a finite face (is not a border edge) the directed edge from vL to $v0$, a *null descriptor* otherwise.
vertex_descriptor	*ep.vR() const*	If $v1v0$ belongs to a finite face (is not a border edge) the third vertex of that triangular face, a *null descriptor* otherwise.
edge_descriptor	*ep.v0_vR() const*	If $v1v0$ belongs to a finite face (is not a border edge) the directed edge from $v0$ to vR, a *null descriptor* otherwise.
edge_descriptor	*ep.vR_v1() const*	If $v1v0$ belongs to a finite face (is not a border edge) the directed edge from vR to $v1$, a *null descriptor* otherwise.

std::vector<vertex_descriptor>

	ep.link() const	The unique sequence of the vertices around $v0v1$ in topological order (*ccw* or *ccw* depending on the relative ordering of *v0* and *v1* in the profile).

std::vector<edge_descriptor>

	ep.border_edges() const	
		The unique collection of the border directed edges incident upon $v0$ and $v1$.
bool	*ep.left_face_exits() const*	
		Indicates if *v0v1* belongs to a finite face of the mesh (i.e, *v0v1* is not a border edge).
bool	*ep.right_face_exits() const*	
		Indicates if *v0v1* belongs to a finite face of the mesh (i.e, *v1v0* is not a border edge).

Has Models

CGAL::Surface_mesh_simplification::Edge_profile<ECM> . . . . page 3304

StopPredicate

Definition

The concept StopPredicate describes the requirements for the predicate which indicates if the simplification process must finish.

Types

StopPredicate:: ECM	The type of the surface to simplify. Must be a model of the *EdgeCollapsableMesh* concept.
StopPredicate:: FT	A field type representing the collapse cost
StopPredicate:: size_type	An integer type representing the number of edges
StopPredicate:: Profile	The type of the edge profile cache. Must be a model of the *EdgeProfile* concept.

Operations

bool *sp.operator()(FT const& current_cost,*
Profile const& profile,
size_type initial_count,
size_type current_count) const

This predicate is called each time an edge is selected for processing, before it is collapsed.
current_cost is the cost of the selected edge.
initial_count and *current_count* are the number of initial and current edges.

If the return value is *true* the simplification terminates before processing the edge, otherwise it continues normally.

Has Models

CGAL::Surface_mesh_simplification::Count_stop_predicate<ECM> . . . . page 3305
CGAL::Surface_mesh_simplification::Count_ratio_stop_predicate<ECM> . . . . page 3306

Functor
Concept

GetCost

Definition

The concept GetCost describes the requirements for the *policy function object* which gets the *collapse cost* of an edge.

The cost returned is a *boost::optional* value (i.e. it can be absent). An *absent* cost indicates that the edge should not be collapsed. This could be the result of a computational limitation (such as overflow), or can be intentionally returned to prevent the edge from being collapsed.

Refines

DefaultConstructible
CopyConstructible

Types

GetCost:: Profile		The type of the edge profile cache. Must be a model of the *EdgeProfile* concept.
GetCost:: FT		A field type representing the collapse cost
typename CGAL::halfedge_graph_traits<ECM>::Point		
	Point;	The point type for the surface vertex. Must be a model of *Point_3*.
boost::optional<FT>	*result_type;*	The type of the result (an optional cost value).

Operations

result_type *gc.operator()(Profile const& edge_profile, boost::optional<Point> const& placement) const*

Computes and returns the cost of collapsing the edge (represented by its profile), using the calculated placement.

Has Models

CGAL::Surface_mesh_simplification::Edge_length_cost<ECM> . . . . page 3307
CGAL::Surface_mesh_simplification::LindstromTurk_cost<ECM> . . . . page 3309

GetPlacement

Functor
Concept

Definition

The concept GetPlacement describes the requirements for the *policy function object* which gets the *collapse placement* of an edge, that is, the new position of the vertex that remains after a halfedge-collapse operation.

The placement returned is a *boost::optional* value (i.e., it can be absent). An absent result indicates that the remaining vertex must be kept in place, not moved to a new position.

Refines

DefaultConstructible
CopyConstructible

Types

GetPlacement:: Profile		The type of the edge profile cache. Must be a model of the *EdgeProfile* concept.
typename CGAL::halfedge_graph_traits<ECM>::Point		
	Point;	The point type for the surface vertex. Must be a model of *Point_3*.
boost::optional<Point>		
	result_type;	The type of the result (an optional point).

Operations

result_type *gp.operator()(Profile const& edge_profile) const*

Computes and returns the placement, that is, the position of the vertex which replaces the collapsing edge (represented by its profile).

Has Models

CGAL::Surface_mesh_simplification::Midpoint_placement<ECM> . . . . page 3308
CGAL::Surface_mesh_simplification::LindstromTurk_placement<ECM> . . . . page 3310

Concept

EdgeCollapseSimplificationVisitor

Definition

The concept EdgeCollapseSimplificationVisitor describes the requirements for the *visitor object* which is used to track the edge collapse simplification algorithm.

The several callbacks given as member functions in the visitor are called from certain places in the algorithm implementation.

Types

EdgeCollapseSimplificationVisitor:: ECM	The type of the surface to simplify. Must be a model of the *EdgeCollapsableMesh* concept.
EdgeCollapseSimplificationVisitor:: FT	A field type representing the collapse cost
EdgeCollapseSimplificationVisitor:: Profile	The type of the edge profile cache. Must be a model of the *EdgeProfile* concept.
typename CGAL::halfedge_graph_traits<ECM>::Point	
Point;	The point type for the surface vertex. Must be a model of *Point_3*.
EdgeCollapseSimplificationVisitor:: size_type	An integer type representing the number of edges

Operations

void *v.OnStarted(ECM& surface)*

Called before the algorithm starts.

void *v.OnFinished(ECM& surface)*

Called after the algorithm finishes.

void *v.OnStopConditionReached(ECM& surface)*

Called when the *StopPredicate* returned *true* (but not if the algorithm terminates because the surface could not be simplified any further).

void *v.OnCollected(Profile profile, boost::optional<FT> cost)*

Called during the *collecting phase* (when a cost is assigned to the edges), for each edge collected.

void *v.OnSelected(Profile profile,*
boost::optional<FT> cost,
size_type initial_count,
size_type current_count)

Called during the *processing phase* (when edges are collapsed), for each edge that is selected.
This method is called before the algorithm checks if the edge is collapsable.
cost indicates the current collapse cost for the edge. If absent (meaning that it could not be computed) the edge will not be collapsed.
initial_count and *current_count* refer to the number of edges.

void *v.OnCollapsing(Profile profile, boost::optional<Point> placement)*

Called when an edge is about to be collapsed and replaced by a vertex whose position is **placement*.
If *placement* is absent (meaning that it could not be computed) the edge will not be collapsed.

void *v.OnNonCollapsable(Profile profile)*

Called for each selected edge which cannot be collapsed because doing so would change the topological type of the surface (turn it into a non-manifold for instance).

Function

CGAL::Surface_mesh_simplification::edge_collapse

Definition

The function *Surface_mesh_simplification::edge_collapse* simplifies in-place a triangulated surface mesh by iteratively collapsing edges.

#include <CGAL/Surface_mesh_simplification/edge_collapse.h>

template<class EdgeCollapsableMesh,class StopPredicate, class P, class T, class R>
int *edge_collapse(EdgeCollapsableMesh& surface,*
StopPredicate should_stop,
sms_named_params<P,T,R> named_parameters)

Simplifies *surface* in-place by collapsing edges, and returns the number of edges effectively removed.

Non-named parameters

surface is the surface to simplify. It must be a model of the *EdgeCollapsableMesh* concept.

should_stop is the stop-condition policy. It must be a model of the *StopPredicate* concept.

Named parameters

named_parameters holds the list of all the additional parameters used by the *edge_collapse* function (including default parameters).

The named parameters list is a composition of function calls separated by a dot (.) where the name of each function matches the name of an argument and wraps the actual parameter.

This is an example with 2 arguments:

vertex_index_map(the_actual_vertex_index_map).edge_index_map(the_actual_edge_index_map)

the_actual_vertex_index_map and *the_actual_edge_index_map* are the actual parameters, while *vertex_index_map()* and *edge_index_map()* are wrapper functions used to designate each formal argument.

All named parameters have default values so you only need to compose those for which the default is inappropriate. Furthermore, since each actual parameter is wrapped in a function whose name designates the formal argument, the order of named parameters in the list is totally irrelevant.

In the following subsections, each named parameter is documented as a helper function. The argument to each helper function is the actual parameter to *edge_collapse()*, while the name of the helper function designates which formal argument it is.

vertex_index_map(VertexIndexMap vpm)

Maps each vertex in the surface into an unsigned integer number in the range *[0,num_vertices(surface))*.

VertexIndexMap must be a ReadablePropertyMap whose *key_type* is *boost::graph_traits<EdgeCollapsableMesh const>::vertex_descriptor* and whose *value_type* is *boost::graph_traits<EdgeCollapsableMesh>::size_type*,

Default: the property map obtained by calling *get(vertex_index,surface)*, which requires the surface vertices to have an *id()* member properly initialized to the required value.
If the vertices don't have such an *id()*, you must pass some property map explicitly. An external property map can be easily obtained by calling *get(vertex_external_index,surface)*. This constructs on the fly, and returns, a property map which non-intrusively associates a proper id with each vertex.

edge_index_map(EdgeIndexMap eim)

Maps each *directed* edge in the surface into an unsigned integer number in the range *[0,num_edges(surface))*.

EdgeIndexMap must be a ReadablePropertyMap whose *key_type* is *boost::graph_traits<EdgeCollapsableMesh const>::edge_descriptor* and whose *value_type* is *boost::graph_traits<EdgeCollapsableMesh>::size_type*

Default: the property map obtained by calling *get(edge_index,surface)*, which requires the surface edges to have an *id()* member properly initialized to the require value.
If the edges don't have such an *id()*, you must pass some property map explicitly. An external property map can be easily obtained by calling *get(edge_external_index,surface)*. This constructs on the fly, and returns, a property map which non-intrusively associates a proper id with each edge.

edge_is_border_map(EdgeIsBorderMap ebm)

Maps each *directed* edge in the surface into a Boolean value which indicates if the edge belongs to the boundary of the surface (facing the outside). *EdgeIsBorderMap* must be a ReadablePropertyMap whose *key_type* is *boost::graph_traits<EdgeCollapsableMesh const>::edge_descriptor* and whose *value_type* is *bool*.

Default: the property map obtained by calling *get(edge_is_border,surface)*.

get_cost(GetCost gc)

The policy which returns the collapse cost for an edge.
The type of *gc* must be a model of the *GetCost* concept.

Default: *CGAL::Surface_mesh_simplification::LindstromTurk_cost<EdgeCollapsableMesh>*.

get_placement(GetPlacement gp)

The policy which returns the placement (position of the replacemet vertex) for an edge.
The type of *gp* must be a model of the *GetPlacement* concept.

Default: *CGAL::Surface_mesh_simplification::LindstromTurk_placement<EdgeCollapsableMesh>*

visitor(EdgeCollapseSimplificationVisitor v)

The visitor that is called by the *edge_collapse* function in certain points to allow the user to track the simplification process.
The type of *v* must be a model of the *EdgeCollapseSimplificationVisitor* concept.

Default: an implementation-defined dummy visitor.

If you wish to provide your own visitor, you can derive from: *CGAL::Surface_mesh_simplification::Edge_collapse_visitor_base*<*EdgeCollapsableMesh*> and override only the callbacks you are interested in.

All these functions naming parameters are defined in *namespace CGAL*. Being non-member functions, they could clash with equally named functions in some other namespace. If that happens, simply qualify the *first* [3] named parameter with *CGAL::*, as shown in the examples in the user manual.

Semantics

The simplification process continues until the *should_stop* policy returns *true* or the surface cannot be simplified any further due to topological constraints.

get_cost and *get_placement* are the policies which control the *cost-strategy*, that is, the order in which edges are collapsed and the remaining vertex is re-positioned.

visitor is used to keep track of the simplification process. It has several member functions which are called at certain points in the simplification code.

[3] The second and subsequent named parameters shall not be qualified as they are member functions

CGAL::Surface_mesh_simplification::Edge_collapse_visitor_base< ECM>

Class

Definition

The class *Surface_mesh_simplification::Edge_collapse_visitor_base<ECM>* provides a base class for models of the *EdgeCollapseSimplificationVisitor* concept. It has one template argument: the type of surface being simplified, which must be a model of the *EdgeCollapsableMesh* concept.

This base class implements all of the visitor's callbacks. This way, users need only override the callbacks they are interested in. The callbacks *are not virtual* because this is not a dynamically polymorphic base class and the derived visitor will never be used polymorphically at runtime (is perfectly fine to override and hide a non-virtual method in the context of the static polymorphism used in the simplification algorithm).

#include <CGAL/Surface_mesh_simplification/Edge_collapse_visitor_base.h>

Is Model for the Concepts

EdgeCollapseSimplificationVisitor

Class

CGAL::Surface_mesh_simplification::Edge_profile<ECM>

Definition

The class *Surface_mesh_simplification::Edge_profile<ECM>* provides a model for the *EdgeProfile* concept. It has one template argument: the type of surface being simplified, which must be a model of the *EdgeCollapsableMesh* concept.

#include <CGAL/Surface_mesh_simplification/Policies/Edge_collapse/Edge_profile.h>

Is Model for the Concepts

EdgeProfile

See Also

GetCost
GetPlacement

CGAL::Surface_mesh_simplification::Count_stop_predicate<ECM>

Functor
Class

Definition

The class *Surface_mesh_simplification::Count_stop_predicate<ECM>* provides a model for the *StopPredicate* concept. It has one template argument: the type of surface being simplified, which must be a model of the *EdgeCollapsableMesh* concept. It returns *true* when the number of current edges drops below a certain threshold.

#include <CGAL/Surface_mesh_simplification/Policies/Edge_collapse/Count_stop_predicate.h>

Is Model for the Concepts

StopPredicate

Creation

Surface_mesh_simplification::Count_stop_predicate<ECM> sp(size_type threshold);

Initializes the predicate establishing the *threshold* value.

Operations

bool *sp.operator()(FT const& current_cost,*
Profile const& edge_profile,
size_type initial_count,
size_type current_count) const

Returns $(current\_count < threshold)$. All other parameters are ignored (but exist since this is a generic policy).

See Also

CGAL::Surface_mesh_simplification::Count_ratio_stop_predicate<ECM>

Functor
Class

CGAL::Surface_mesh_simplification::Count_ratio_stop_predicate< ECM>

Definition

The class *Surface_mesh_simplification::Count_ratio_stop_predicate<ECM>* provides a model for the *StopPredicate* concept. It has one template argument: the type of surface being simplified, which must be a model of the *EdgeCollapsableMesh* concept. It returns *true* when the relation between the initial and current number of edges drops below a certain ratio.

#include <CGAL/Surface_mesh_simplification/Policies/Edge_collapse/Count_ratio_stop_predicate.h>

Is Model for the Concepts

StopPredicate

Creation

Surface_mesh_simplification::Count_ratio_stop_predicate<ECM> sp(double ratio);

Initializes the predicate establishing the *ratio*.

Operations

bool *sp.operator()(FT const& current_cost, Profile const& edge_profile, size_type initial_count, size_type current_count) const*

Returns $(((double)current\_count/(double)initial\_count) < ratio)$. All other parameters are ignored (but exist since this is a generic policy).

See Also

CGAL::Surface_mesh_simplification::Count_stop_predicate<ECM>

CGAL::Surface_mesh_simplification::Edge_length_cost<ECM>

Functor
Class

Definition

The class *Surface_mesh_simplification::Edge_length_cost<ECM>* provides a model for the *GetCost* concept. It has one template argument: the type of surface being simplified, which must be a model of the *EdgeCollapsableMesh* concept. It computes the collapse cost as the squared length of the edge.

#include <CGAL/Surface_mesh_simplification/Policies/Edge_collapse/Edge_length_cost.h>

Is Model for the Concepts

GetCost

Creation

Surface_mesh_simplification::Edge_length_cost<ECM> gc;

Default constructor

Operations

result_type *gc.operator()(Profile const& profile, boost::optional<Point> const& placement) const*

Returns the *collapse cost* as the squared distance between the points of the source and target vertices (that is, *profile.p0()* and *profile.p1()*.
The *placement* argument is ignored.

Functor
Class

CGAL::Surface_mesh_simplification::Midpoint_placement<ECM>

Definition

The class *Surface_mesh_simplification::Midpoint_placement<ECM>* provides a model for the *GetPlacement* concept. It computes the placement as the midpoint position along the edge.

The class *Surface_mesh_simplification::Midpoint_placement<ECM>* has one template arguments: the type of surface being simplified. It be a model of the *EdgeCollapsableMesh* concept.

#include <CGAL/Surface_mesh_simplification/Policies/Edge_collapse/Midpoint_placement.h>

Is Model for the Concepts

GetPlacement

Creation

Surface_mesh_simplification::Midpoint_placement<ECM> gp;

Default constructor

Operations

result_type *gp.operator()(Profile const& edge_profile) const*

Returns the *placement* (vertex position) as the midpoint between the points of the source and target vertices (*profile.p0()* and *profile.p1()*)

CGAL::Surface_mesh_simplification::LindstromTurk_cost<ECM>

Functor
Class

Definition

The class *Surface_mesh_simplification::LindstromTurk_cost<ECM>* provides a model for the *GetCost* concept. It computes the collapse cost following the Lindstrom-Turk strategy (Section 53.3.1)

The class *Surface_mesh_simplification::LindstromTurk_cost<ECM>* has one template argument: the type of surface being simplified. It must be a model of the *EdgeCollapsableMesh* concept.

#include <CGAL/Surface_mesh_simplification/Policies/Edge_collapse/LindstromTurk_cost.h>

Is Model for the Concepts

GetCost

Creation

Surface_mesh_simplification::LindstromTurk_cost<ECM> gc(FT factor = FT(0.5));

Initializes the policy with the given *weighting unit factor*. See 53.3.1 for details on the meaning of this factor.

Operations

result_type *gc.operator()(Profile const& profile, boost::optional<Point> const& placement) const*

Returns the cost of collapsing the edge (represented by its profile) considering the new *placement* computed for it.

See Also

CGAL::Surface_mesh_simplification::LindstromTurk_placement<ECM>

CGAL::Surface_mesh_simplification::LindstromTurk_placement< ECM>

Definition

The class *Surface_mesh_simplification::LindstromTurk_placement<ECM>* provides a model for the *GetPlacement* concept. It computes the placement, that is, the new position for the remaining vertex after a halfedge-collapse, following the Lindstrom-Turk strategy (Section 53.3.1).

The class *Surface_mesh_simplification::LindstromTurk_placement<ECM>* has one template argument: the type of surface being simplified. It must be a model of the *EdgeCollapsableMesh* concept.

#include <CGAL/Surface_mesh_simplification/Policies/Edge_collapse/LindstromTurk_placement.h>

Is Model for the Concepts

GetPlacement

Creation

Surface_mesh_simplification::LindstromTurk_placement<ECM> gp(FT factor = FT(0.5));

Initializes the policy with the given *weighting unit factor*. See 53.3.1 for details on the meaning of this factor.

Operations

result_type *gp.operator()(Profile const& edge_profile) const*

Returns the new position for the remaining vertex after collapsing the edge (represented by its profile).

See Also

CGAL::Surface_mesh_simplification::LindstromTurk_cost<ECM>

Chapter 54

Planar Parameterization of Triangulated Surface Meshes

Laurent Saboret, Pierre Alliez and Bruno Lévy

Contents

54.1 Introduction

Parameterizing a surface amounts to finding a one-to-one mapping from a suitable domain to the surface. A good mapping is the one which minimizes either angle distortions (conformal parameterization) or area distortions (equiareal parameterization) in some sense. In this package, we focus on parameterizing triangulated surfaces which are homeomorphic to a disk, and on piecewise linear mappings onto a planar domain.

Although the main motivation behind the first parameterization methods was the application to texture mapping, it is now frequently used for mapping more sophisticated modulation signals (such as normal, transparency, reflection or light modulation maps), fitting scattered data, re-parameterizing spline surfaces, repairing CAD models, approximating surfaces and remeshing.

This CGAL package implements some of the state-of-the-art surface parameterization methods, such as least squares conformal maps, discrete conformal map, discrete authalic parameterization, Floater mean value coordinates or Tutte barycentric mapping. These methods mainly distinguish by the distortion they minimize (angles vs. areas), by the constrained border onto the planar domain (convex polygon vs. free border) and by the guarantees provided in terms of bijective mapping.

The package proposes currently an interface with *CGAL::Polyhedron_3<Traits>* data structure.

Since parameterizing meshes require efficient representation of sparse matrices and efficient iterative or direct linear solvers, we provide a unified interface to a linear solver library (EIGEN), and propose a separate package devoted to OpenNL sparse linear solver.

Note that linear solvers commonly use double precision floating point numbers. Therefore, this package is intended to be used with a CGAL Cartesian kernel with doubles.

The intended audience of this package is researchers, developers or students developing algorithms around parameterization of triangle meshes for geometry processing as well as for signal mapping on triangulated surfaces.

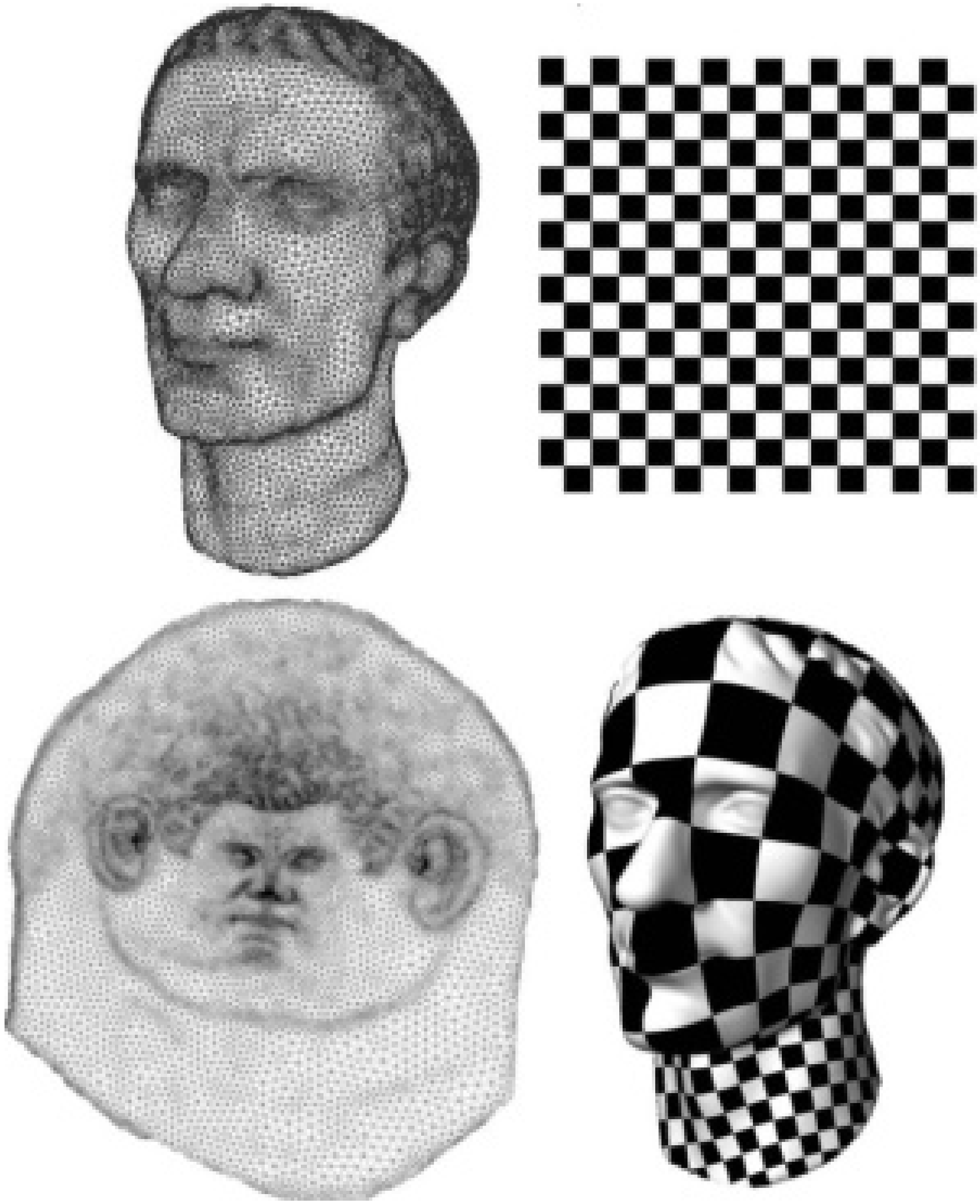


Figure 54.1: Texture mapping via Least Squares Conformal Maps parameterization. Top: original mesh and texture. Bottom: parameterized mesh (left: parameter space, right: textured mesh).

54.2 Basics

54.2.1 Default Surface Parameterization

From the user point of view, the simplest entry point to this package is the following function:

Parameterizer_traits_3<ParameterizationMesh_3>::Error_code

parameterize(ParameterizationMesh_3 & mesh)

Compute a one-to-one mapping from a 3D triangle surface mesh to a 2D circle, using Floater Mean Value Coordinates algorithm. A one-to-one piecewise linear mapping is guaranteed. The result is a pair of (u,v) parameter coordinates for each vertex of the input mesh.
Preconditions: mesh must be a triangle mesh surface with one connected component.

The function *CGAL::parameterize()* applies a default surface parameterization method: Floater Mean Value Coordinates [Flo03a], with an arc-length circular border parameterization, and using OpenNL sparse linear solver [Lev05]. The *ParameterizationMesh_3* concept defines the input meshes handled by *CGAL::parameterize()*. See Section 54.2.2. The result is stored into the (u,v) fields of the mesh halfedges.

Note: *CGAL::Parameterizer_traits_3<ParameterizationMesh_3>* is the (pure virtual) superclass of all surface parameterizations and defines the error codes.

54.2.2 Input Mesh for parameterize()

The input meshes handled *directly* by *CGAL::parameterize()* must be models of *ParameterizationMesh_3*, triangulated, 2-manifold, oriented, and homeomorphic to discs (possibly with holes).

Note: *ParameterizationMesh_3* is a general concept to access a polyhedral mesh. It is optimized for the *Surface_mesh_parameterization* package only in the sense that it defines the accessors to fields specific to the parameterization domain (*index*, *u*, *v*, *is_parameterized*). The extra constraints needed by the surface parameterization methods (triangulated, 2-manifold, homeomorphic to a disc) are not part of the concept and are checked at runtime.

This package provides a model of the *ParameterizationMesh_3* concept to access *CGAL::Polyhedron_3<Traits>* :
CGAL::Parameterization_polyhedron_adaptor_3<Polyhedron_3_>

We will see later that *CGAL::parameterize()* can support *indirectly* meshes that are not topological disks.

54.2.3 Default Parameterization Example

Simple_parameterization.cpp applies the default parameterization to a *CGAL::Polyhedron_3<Traits>* mesh (must be a topological disk). Eventually, it extracts the result from halfedges and prints it.

```
#include <CGAL/Cartesian.h>
#include <CGAL/Polyhedron_3.h>
```

```
#include <CGAL/IO/Polyhedron_iostream.h>
#include <CGAL/Parameterization_polyhedron_adaptor_3.h>
#include <CGAL/parameterize.h>

#include <iostream>
#include <fstream>


//
    ----------------------------------------------------------------------------
// Private types
//
    ----------------------------------------------------------------------------

typedef CGAL::Cartesian<double>             Kernel;
typedef CGAL::Polyhedron_3<Kernel>          Polyhedron;


//
    ----------------------------------------------------------------------------
// main()
//
    ----------------------------------------------------------------------------

int main(int argc, char * argv[])
{
    std::cerr << "PARAMETERIZATION" << std::endl;
    std::cerr << "  Floater parameterization" << std::endl;
    std::cerr << "  Circle border" << std::endl;
    std::cerr << "  OpenNL solver" << std::endl;

    //***************************************
    // decode parameters
    //***************************************

    if (argc-1 != 1)
    {
        std::cerr << "Usage: " << argv[0] << " input_file.off" <<
   std::endl;
        return(EXIT_FAILURE);
    }

    // File name is:
    const char* input_filename  = argv[1];

    //***************************************
    // Read the mesh
    //***************************************

    // Read the mesh
    std::ifstream stream(input_filename);
    Polyhedron mesh;
    stream >> mesh;
    if(!stream || !mesh.is_valid() || mesh.empty())
```

```
 {
      std::cerr << "Error: cannot read OFF file " << input_filename <<
std::endl;
      return EXIT_FAILURE;
 }

 //***************************************
 // Create Polyhedron adaptor
 // Note: no cutting => we support only
 // meshes that are topological disks
 //***************************************

 typedef CGAL::Parameterization_polyhedron_adaptor_3<Polyhedron>

Parameterization_polyhedron_adaptor;
 Parameterization_polyhedron_adaptor mesh_adaptor(mesh);

 //***************************************
 // Floater Mean Value Coordinates parameterization
 // (defaults are circular border and OpenNL solver)
 //***************************************

 typedef
CGAL::Parameterizer_traits_3<Parameterization_polyhedron_adaptor>
                                         Parameterizer;  // Type that
defines the error codes

 Parameterizer::Error_code err = CGAL::parameterize(mesh_adaptor);
 switch(err) {
 case Parameterizer::OK: // Success
     break;
 case Parameterizer::ERROR_EMPTY_MESH: // Input mesh not supported
 case Parameterizer::ERROR_NON_TRIANGULAR_MESH:
 case Parameterizer::ERROR_NO_TOPOLOGICAL_DISC:
 case Parameterizer::ERROR_BORDER_TOO_SHORT:
     std::cerr << "Input mesh not supported: " <<
Parameterizer::get_error_message(err) << std::endl;
     return EXIT_FAILURE;
     break;
 default: // Error
     std::cerr << "Error: " << Parameterizer::get_error_message(err)
<< std::endl;
     return EXIT_FAILURE;
     break;
 };

 //***************************************
 // Output
 //***************************************

 // Raw output: dump (u,v) pairs
 Polyhedron::Vertex_const_iterator pVertex;
 for (pVertex = mesh.vertices_begin();
     pVertex != mesh.vertices_end();
```

```
        pVertex++)
    {
        // (u,v) pair is stored in any halfedge
        double u = mesh_adaptor.info(pVertex->halfedge())->uv().x();
        double v = mesh_adaptor.info(pVertex->halfedge())->uv().y();
        std::cout << "(u,v) = (" << u << "," << v << ")" << std::endl;
    }

    return EXIT_SUCCESS;
}
```

File: examples/Surface_mesh_parameterization/Simple_parameterization.cpp

54.2.4 Enhanced parameterize() function

This package provides a second *CGAL::parameterize()* entry point where the user can specify a parameterization method:

Parameterizer_traits_3<ParameterizationMesh_3>::Error_code

parameterize(ParameterizationMesh_3 & mesh,
ParameterizerTraits_3 parameterizer)

Compute a one-to-one mapping from a 3D triangle surface 'mesh' to a simple 2D domain. The mapping is piecewise linear on the triangle mesh. The result is a pair (u,v) of parameter coordinates for each vertex of the input mesh. One-to-one mapping may be guaranteed or not, depending on the chosen *ParametizerTraits_3* algorithm.
Preconditions: 'mesh' must be a triangle surface mesh with one connected component, and the mesh border must be mapped onto a convex polygon (for fixed border parameterizations).

54.2.5 Introduction to the Package Concepts

The ParameterizerTraits_3 concept

This CGAL package implements some of the state-of-the-art surface parameterization methods, such as Least Squares Conformal Maps, Discrete Conformal Map, Discrete Authalic Parameterization, Floater Mean Value Coordinates or Tutte Barycentric Mapping. These methods are provided as models of the *ParameterizerTraits_3* concept. See Section 54.3.

Each of these surface parameterization methods is templated by the input mesh type, a border parameterization and a solver:

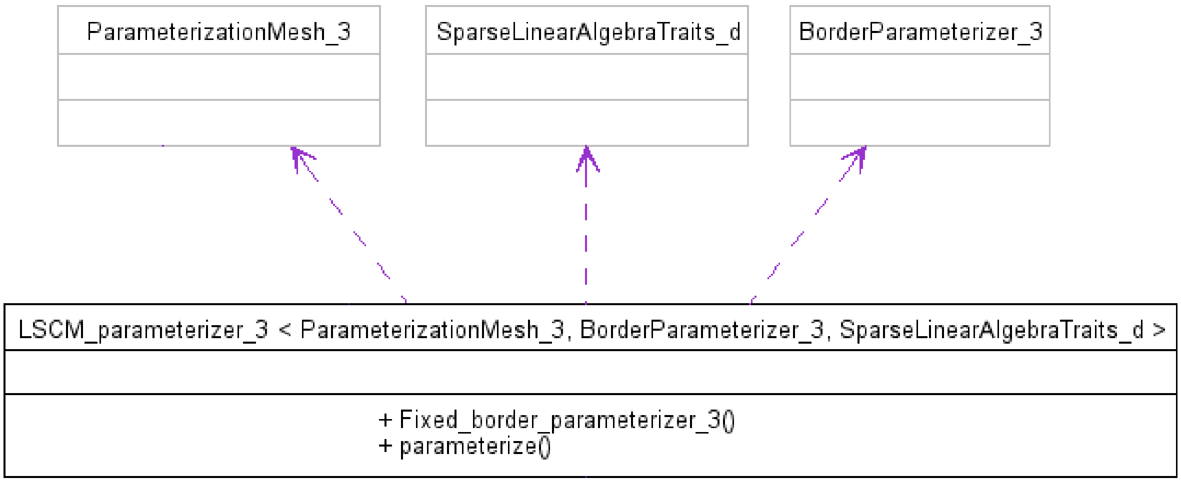


Figure 54.2: A parameterizer UML class diagram (simplified).

The BorderParameterizer_3 concept

Parameterization methods for borders are used as traits classes modifying the behavior of *ParameterizerTraits_3* models. They are provided as models of the *BorderParameterizer_3* concept. See Sections 54.3.1 and 54.3.2.

The SparseLinearAlgebraTraits_d concept

This package solves sparse linear systems using solvers which are models of *SparseLinearAlgebraTraits_d*. See Section 54.4.

The ParameterizationMesh_3 and ParameterizationPatchableMesh_3 Concepts

As described in Section 54.2.2 the input meshes handled by *CGAL::parameterize()* must be models of the *ParameterizationMesh_3* concept. The surface parameterization methods provided by this package only support surfaces which are homeomorphic to disks, possibly with holes. Nevertheless meshed with arbitrary topology and number of connected components can be parameterized, provided that the user specifies a *cut graph* (an oriented list of vertices) which is the border of a topological disc. If no cut graph is specified as input, the longest border of the input mesh is taken by default, the others being considered as holes.

For this purpose, the *CGAL::Parameterization_mesh_patch_3<ParameterizationPatchableMesh_3>* class is responsible for *virtually* cutting a patch into a *ParameterizationPatchableMesh_3* mesh. The resulting patch is a topological disk (if the input cutting path is correct) and provides a *ParameterizationMesh_3* interface. It can be used as parameter for the function *CGAL::parameterize()*.

ParameterizationPatchableMesh_3 inherits from *ParameterizationMesh_3*, thus is a concept for a 3D surface mesh. *ParameterizationPatchableMesh_3* adds the ability to support patches and virtual seams. *Patches* are a subset of a 3D mesh. *Virtual seams* behave as if the surface was cut along a cut graph. More information is provided in Section 54.5.

54.3 Surface Parameterization Methods

This CGAL package implements some of the state-of-the-art surface parameterization methods, such as Least Squares Conformal Maps, Discrete Conformal Map, Discrete Authalic Parameterization, Floater Mean Value Coordinates or Tutte Barycentric Mapping. These methods are provided as models of the *ParameterizerTraits_3* concept.

54.3.1 Fixed Border Surface Parameterizations

Fixed Border Surface Parameterizations need a set of constraints: two (u,v) coordinates for each vertex along the border. Such border parameterizations are described in Section 54.3.1.

Tutte Barycentric Mapping

CGAL::Barycentric_mapping_parameterizer_3<ParameterizationMesh_3, BorderParameterizer_3, SparseLinearAlgebraTraits_d>

The Barycentric Mapping parameterization method has been introduced by Tutte [Tut63]. In parameter space, each vertex is placed at the barycenter of its neighbors to achieve the so-called convex combination condition. This algorithm amounts to solve one sparse linear solver for each set of parameter coordinates, with a #vertices x #vertices sparse and symmetric positive definite matrix (if the border vertices are eliminated from the linear system). A coefficient (i, j) of the matrix is set to 1 for an edge linking the vertex v_i to the vertex v_j, to minus the degree of the vertex v_i for a diagonal element, and to 0 for any other matrix entry. Although a bijective mapping is guaranteed when the border is convex, this method does not minimize angles nor areas distortion.

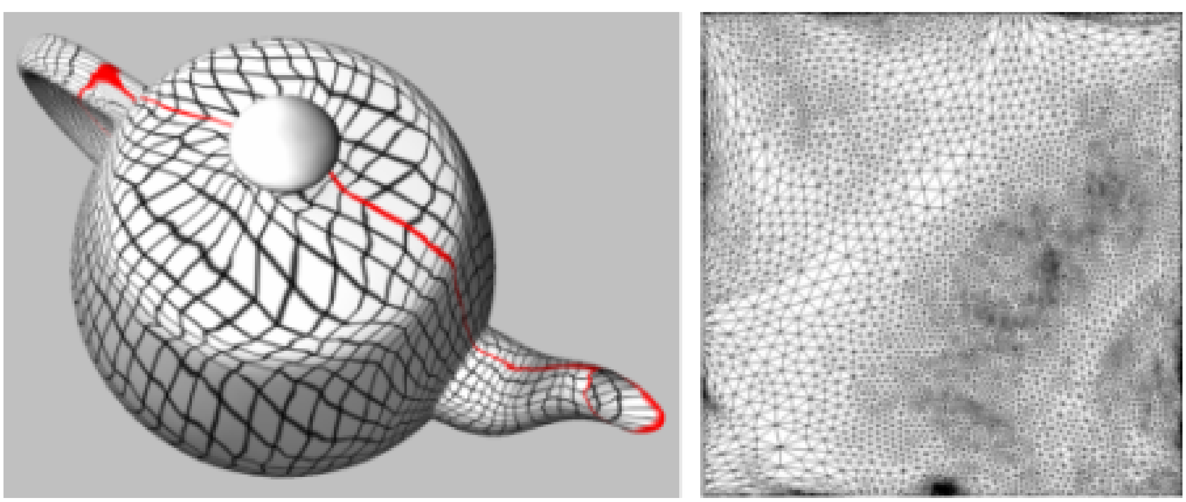


Figure 54.3: Left: Tutte barycentric mapping parameterization (the red line depicts the cut graph). Right: parameter space.

Discrete Conformal Map

CGAL::Discrete_conformal_map_parameterizer_3<ParameterizationMesh_3, BorderParameterizer_3, SparseLinearAlgebraTraits_d>

Discrete conformal map parameterization has been introduced by Eck et al. to the graphics community [EDD+95]. It attempts to lower angle deformation by minimizing a discrete version of the Dirichlet energy as derived by Pinkall and Polthier [PP93]. A one-to-one mapping is guaranteed only when the two following conditions are fulfilled: the barycentric mapping condition (each vertex in parameter space is a convex combination if its neighboring vertices), and the border is convex. This method solves two #vertices x #vertices sparse linear systems. The matrix (the same for both systems) is sparse and symmetric definite positive (if the border vertices are eliminated from the linear system and if the mesh contains no hole), thus can be efficiently solved using dedicated linear solvers.

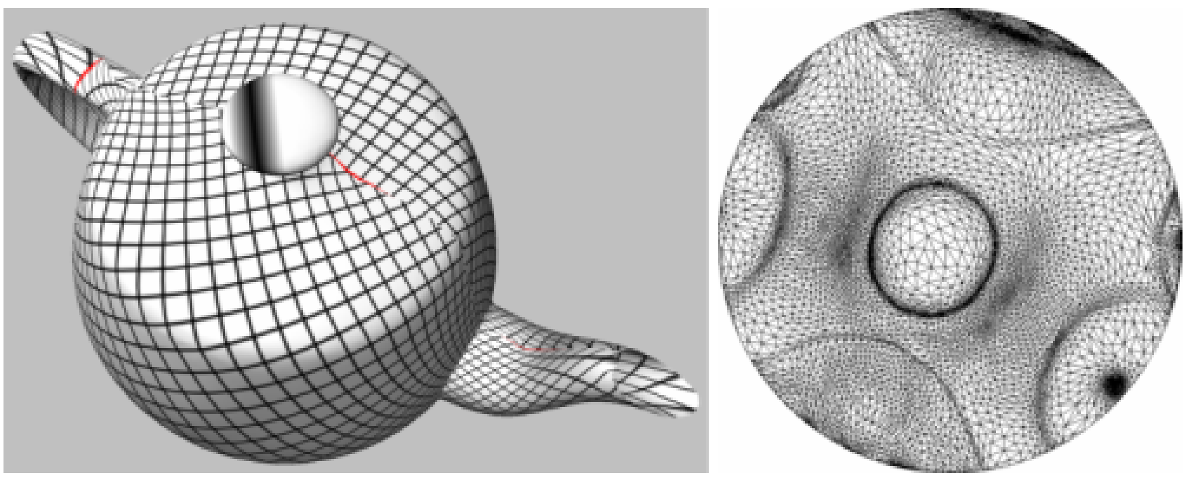


Figure 54.4: Left: discrete conformal map. Right: parameter space.

Floater Mean Value Coordinates

CGAL::Mean_value_coordinates_parameterizer_3<ParameterizationMesh_3, BorderParameterizer_3, SparseLinearAlgebraTraits_d>

The mean value coordinates parameterization method has been introduced by Floater [Flo03a]. Each vertex in parameter space is optimized so as to be a convex combination of its neighboring vertices. The barycentric coordinates are this time unconditionally positive, by deriving an application of the mean theorem for harmonic functions. This method is in essence an approximation of the discrete conformal maps, with a guaranteed one-to-one mapping when the border is convex. This method solves two #vertices x #vertices sparse linear systems. The matrix (the same for both systems) is asymmetric.

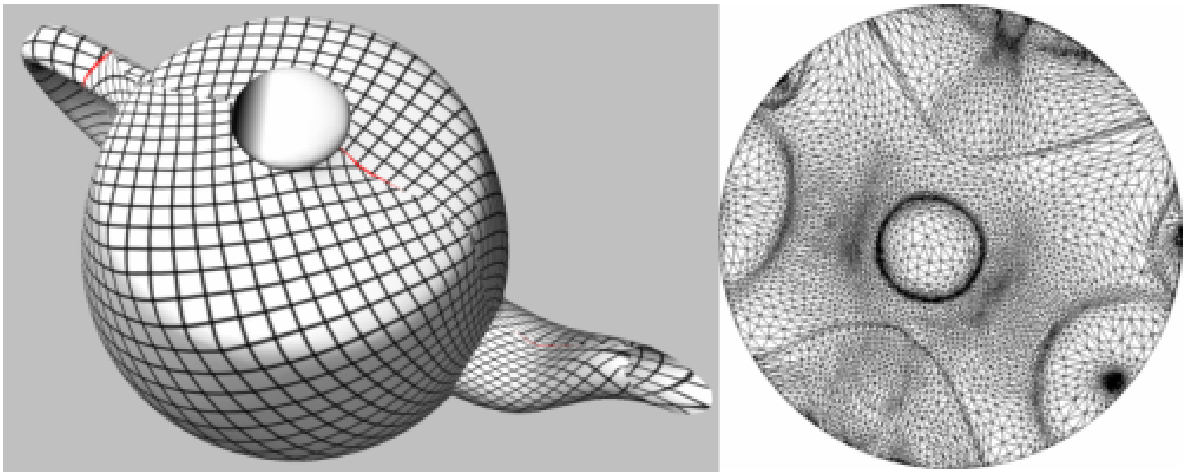


Figure 54.5: Floater Mean Value Coordinates

Discrete Authalic parameterization

CGAL::Discrete_authalic_parameterizer_3<ParameterizationMesh_3, BorderParameterizer_3, SparseLinearAlgebraTraits_d>

The discrete authalic parameterization method has been introduced by Desbrun et al. [DMA02]. It corresponds to a weak formulation of an area-preserving method, and in essence locally minimizes the area distortion. A one-to-one mapping is guaranteed only if the convex combination condition is fulfilled and the border is convex. This method solves two #vertices x #vertices sparse linear systems. The matrix (the same for both systems) is asymmetric.

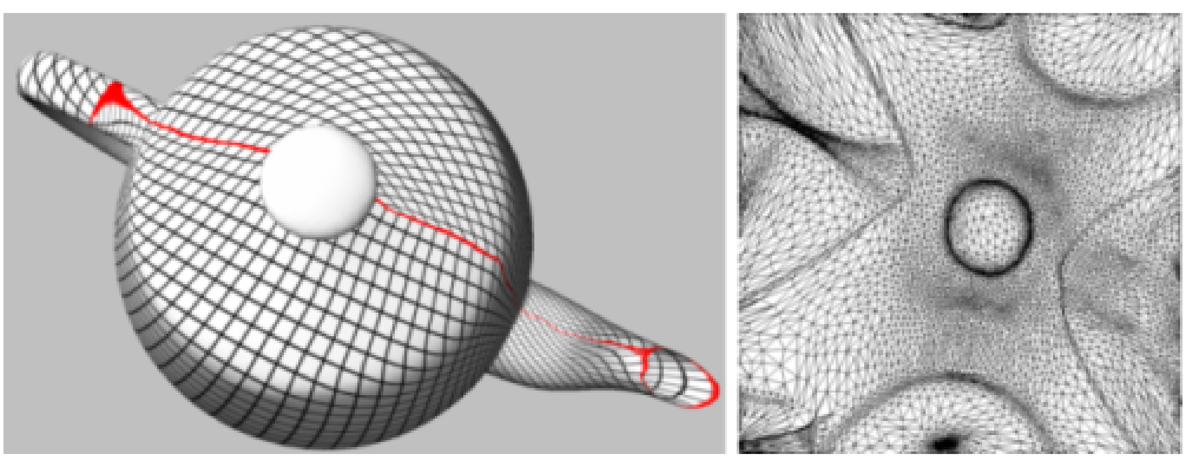


Figure 54.6: Discrete Authalic Parameterization

Border Parameterizations for Fixed Methods

Parameterization methods for borders are used as traits classes modifying the behavior of *ParameterizerTraits_3* models. They are provided as models of the *BorderParameterizer_3* concept. Border parameterizations for fixed border surface parameterizations are a family of methods to define a set of constraints, namely two u, v coordinates for each vertex along the border.

- The user can select a border parameterization among two commonly used methods: uniform or arc-length parameterization.

  Usage:

  Uniform border parameterization is more stable, although it gives poor visual results. The arc-length border parameterization is used by default.

- One convex shape specified by one shape among two standard ones: a circle or a square.

  Usage:

  The circular border parameterization is used by default as it corresponds to the simplest convex shape. The square border parameterization is commonly used for texture mapping.

CGAL::Circular_border_arc_length_parameterizer_3<ParameterizationMesh_3>
CGAL::Circular_border_uniform_parameterizer_3<ParameterizationMesh_3>
CGAL::Square_border_arc_length_parameterizer_3<ParameterizationMesh_3>
CGAL::Square_border_uniform_parameterizer_3<ParameterizationMesh_3>

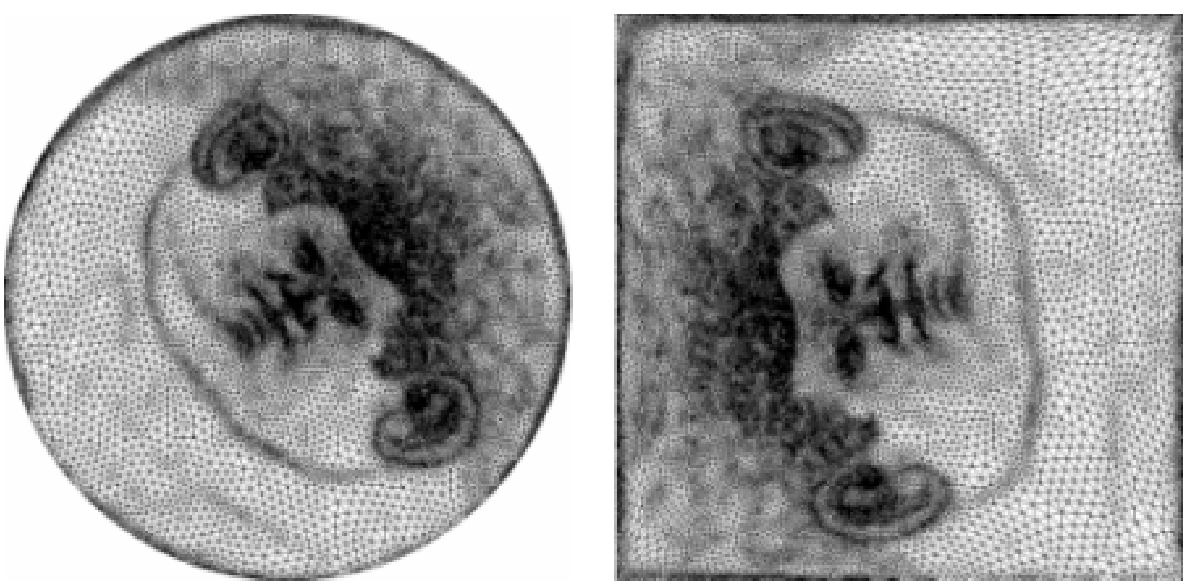


Figure 54.7: Left: Julius Cesar mask parameterization with Authalic/circular border. Right: Julius Cesar mask's image with Floater/square border.

54.3.2 Free Border Surface Parameterizations

Least Squares Conformal Maps

CGAL::LSCM_parameterizer_3<ParameterizationMesh_3, BorderParameterizer_3, SparseLinearAlgebraTraits_d>

The Least Squares Conformal Maps (LSCM) parameterization method has been introduced by Lévy et al. [LPRM02]. It corresponds to a conformal method with a free border (at least two vertices have to be constrained to obtain a unique solution), which allows further lowering of the angle distortion. A one-to-one mapping is not guaranteed by this method. It solves a (2 × #triangles) × #vertices sparse linear system in the least squares sense, which implies solving a symmetric matrix.

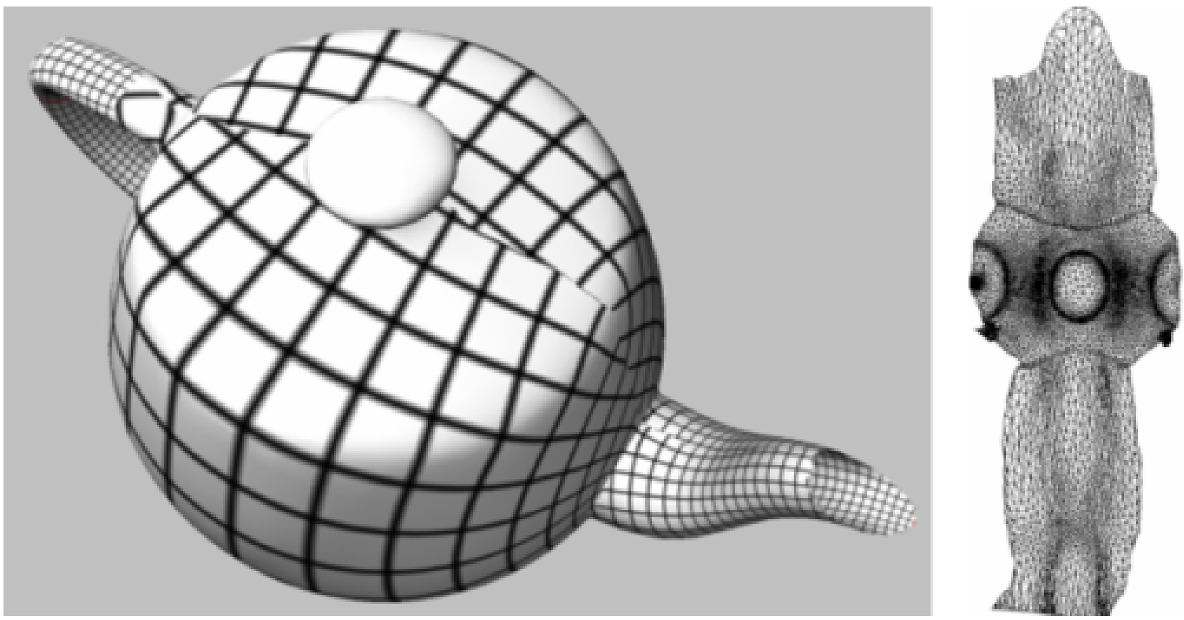


Figure 54.8: Least squares conformal maps.

Border Parameterizations for Free Methods

Parameterization methods for borders are used as traits classes modifying the behavior of *ParameterizerTraits_3* models. They are provided as models of the *BorderParameterizer_3* concept. The border parameterizations associated to free border surface parameterization methods define only two constraints: the pinned vertices.

- *CGAL::Two_vertices_parameterizer_3<ParameterizationMesh_3>*

  Usage:

  CGAL::Two_vertices_parameterizer_3<ParameterizationMesh_3> is the default free border parameterization, and is the only one available in the current version of this package.

54.3.3 Discrete Authalic Parameterization Example

Authalic_parameterization.cpp computes a Discrete Authalic parameterization over a *CGAL::Polyhedron_3< Traits>* mesh. Specifying a specific surface parameterization instead of the default one means using the second parameter of *CGAL::parameterize()*. The differences with the first example *Simple_parameterization.cpp* are:

```
#include <CGAL/Discrete_authalic_parameterizer_3.h>

...

//***************************************
// Discrete Authalic Parameterization
//***************************************

typedef
   CGAL::Discrete_authalic_parameterizer_3<Parameterization_polyhedron_adaptor>
                                                    Parameterizer;

Parameterizer::Error_code err = CGAL::parameterize(mesh_adaptor,
   Parameterizer());

...
```

54.3.4 Square Border Arc Length Parameterization Example

Square_border_parameterization.cpp computes a Floater mean value coordinates parameterization with a square border arc length parameterization. Specifying a specific border parameterization instead of the default one means using the second parameter of *CGAL::Mean_value_coordinates_parameterizer_3< ParameterizationMesh_3, BorderParameterizer_3, SparseLinearAlgebraTraits_d>*. The differences with the first example *Simple_parameterization.cpp* are:

```
#include <CGAL/Square_border_parameterizer_3.h>

...

//***************************************
// Floater Mean Value Coordinates parameterization
// with square border
//***************************************

// Square border parameterizer
typedef
   CGAL::Square_border_arc_length_parameterizer_3<Parameterization_polyhedron_adaptor>

   Border_parameterizer;

// Floater Mean Value Coordinates parameterizer with square border
typedef
   CGAL::Mean_value_coordinates_parameterizer_3<Parameterization_polyhedron_adaptor,

   Border_parameterizer>
```

```
                                                    Parameterizer;

Parameterizer::Error_code err = CGAL::parameterize(mesh_adaptor,
    Parameterizer());

...
```

54.4 Sparse Linear Algebra

Parameterizing triangle meshes requires both efficient representation of sparse matrices and efficient iterative or direct linear solvers. We provide links to EIGEN library and include a separate package devoted to OpenNL sparse linear solver.

54.4.1 List of Solvers

We provide an interface to several sparse linear solvers, as models of the *SparseLinearAlgebraTraits_d* concept:

- An interface to sparse solvers from the OPENNL library [Lev05] is provided through classes *OpenNL::DefaultLinearSolverTraits<COEFFTYPE, MATRIX, VECTOR, SOLVER>* and *OpenNL::SymmetricLinearSolverTraits<COEFFTYPE, MATRIX, VECTOR, SOLVER>*. The OpenNL library version shipped with CGAL is a lightweight default sparse linear solver. It does not support large systems, but it is portable and supports exact number types.
- An interface to all sparse solvers from the Eigen (Page 28) library is provided through the class *CGAL::Eigen_solver_traits<T>*. This solver traits class can be used for an iterative or a direct, symmetric or general sparse solvers. The EIGEN solver to be used must be given as template parameter.

54.4.2 EIGEN Solver Example

The example *examples/Surface_mesh_parameterization/Eigen_parameterization.cpp* computes the default parameterization method (Floater mean value coordinates with a circular border), but specifically instantiates an EIGEN solver. Specifying a specific solver instead of the default one (OpenNL) means using the third parameter of *CGAL::Mean_value_coordinates_parameterizer_3<ParameterizationMesh_3, BorderParameterizer_3, SparseLinearAlgebraTraits_d>*. The differences with the first example *examples/Surface_mesh_parameterization/Simple_parameterization.cpp* are:

```
#include <CGAL/Eigen_solver_traits.h>

...

//***************************************
// Floater Mean Value Coordinates parameterization
// (circular border) with Eigen solver
//***************************************

// Circular border parameterizer (the default)
```

```
typedef
    CGAL::Circular_border_arc_length_parameterizer_3<Parameterization_polyhedron_adaptor>
                                                              Border_parameterizer;
// Eigen solver
typedef CGAL::Eigen_solver_traits<>                          Solver;

// Floater Mean Value Coordinates parameterization
// (circular border) with Eigen solver
typedef
    CGAL::Mean_value_coordinates_parameterizer_3<Parameterization_polyhedron_adaptor,

    Border_parameterizer,
                                                                   Solver>
                                                              Parameterizer;

Parameterizer::Error_code err = CGAL::parameterize(mesh_adaptor,
    Parameterizer());

...
```

54.5 Cutting a Mesh

54.5.1 Computing a Cut Graph

All surface parameterization methods proposed in this package only deal with meshes which are homeomorphic (topologically equivalent) to discs. Nevertheless meshes with arbitrary topology and number of connected components car be parameterized, provided that the user specifies a cut graph (an oriented list of vertices), which is the border of a topological disc. If no cut graph is provided as input, the longest border already in the input mesh is taken as default border, all other borders being considered as holes. Note that only the inside part (i.e., one connected component) of the given border is parameterized.

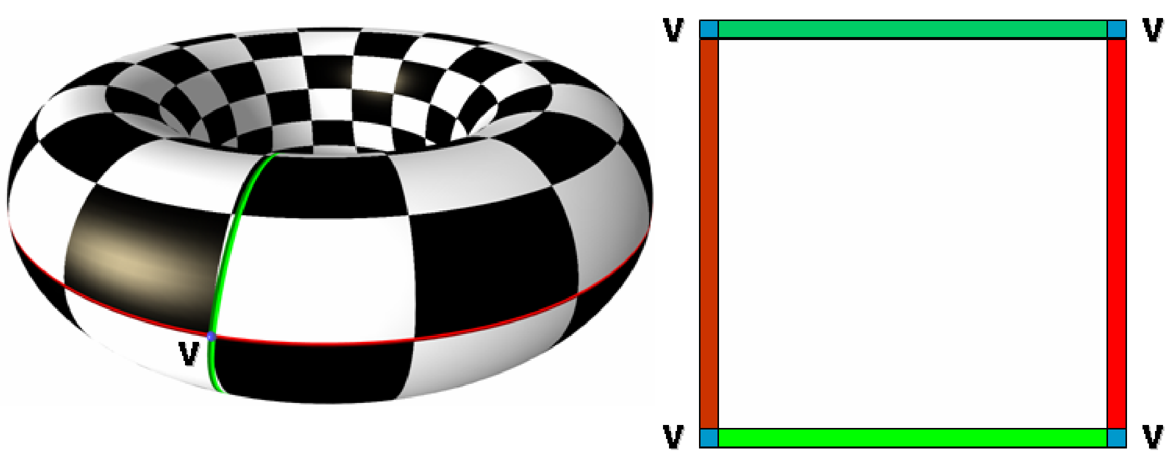


Figure 54.9: Cut Graph

This package does not provide any algorithm to transform an arbitrary mesh into a topological disk, the user being responsible for generating such a cut graph. Nevertheless, we provide in *polyhedron_ex_parameterization.cpp* a simple cutting algorithm for the sake of completeness.

54.5.2 Applying a Cut

The surface parameterization classes in this package only *directly* support surfaces which are homeomorphic to disks (models of *ParameterizationMesh_3*). This software design simplifies the implementation of all new parameterization methods.

The *CGAL::Parameterization_mesh_patch_3<ParameterizationPatchableMesh_3>* class is responsible for *virtually* cutting a patch in a *ParameterizationPatchableMesh_3* mesh. The resulting patch is a topological disk (if the cut graph is correct) and provides a *ParameterizationMesh_3* interface. It can be used as parameter of *CGAL::parameterize()*.

ParameterizationPatchableMesh_3 inherits from concept *ParameterizationMesh_3*, thus is a concept for a 3D surface mesh. *ParameterizationPatchableMesh_3* adds the ability to support patches and virtual seams. *Patches* are a subset of a 3D mesh. *Virtual seams* behave exactly as if the surface was cut along a certain graph.

The *ParameterizationMesh_3* interface with the Polyhedron is both a model of *ParameterizationMesh_3* and *ParameterizationPatchableMesh_3*:
CGAL::Parameterization_polyhedron_adaptor_3<Polyhedron_3_>

Note that this class is a decorator which adds *on the fly* the necessary fields to unmodified CGAL data structures (using STL maps). For better performances, it is recommended to use CGAL data structures enriched with the proper fields. See *Polyhedron_ex* class in *polyhedron_ex_parameterization.cpp* example.

54.5.3 Cutting a Mesh Example

Mesh_cutting_parameterization.cpp virtually cuts a *CGAL::Polyhedron_3<Traits>* mesh to make it a topological disk, then applies the default parameterization:

```
#include <CGAL/Cartesian.h>
#include <CGAL/Polyhedron_3.h>
#include <CGAL/IO/Polyhedron_iostream.h>
#include <CGAL/Parameterization_polyhedron_adaptor_3.h>
#include <CGAL/parameterize.h>
#include <CGAL/Parameterization_mesh_patch_3.h>

#include <iostream>
#include <fstream>


//
    ----------------------------------------------------------------------------
// Private types
//
    ----------------------------------------------------------------------------

typedef CGAL::Cartesian<double>             Kernel;
typedef CGAL::Polyhedron_3<Kernel>          Polyhedron;
```

```
// Polyhedron adaptor
typedef CGAL::Parameterization_polyhedron_adaptor_3<Polyhedron>

   Parameterization_polyhedron_adaptor;

// Type describing a border or seam as a vertex list
typedef std::list<Parameterization_polyhedron_adaptor::Vertex_handle>
                                                    Seam;


//
   ----------------------------------------------------------------------------
// Private functions
//
   ----------------------------------------------------------------------------

// If the mesh is a topological disk, extract its longest border,
// else compute a very simple cut to make it homeomorphic to a disk.
// Return the border of this region (empty on error)
//
// CAUTION: this cutting algorithm is very naive. Write your own!
static Seam cut_mesh(Parameterization_polyhedron_adaptor& mesh_adaptor)
{
    // Helper class to compute genus or extract borders
    typedef
   CGAL::Parameterization_mesh_feature_extractor<Parameterization_polyhedron_adaptor>
                                                    Mesh_feature_extractor;

    Seam seam;              // returned list

    // Get reference to Polyhedron_3 mesh
    Polyhedron& mesh = mesh_adaptor.get_adapted_mesh();

    // Extract mesh borders and compute genus
    Mesh_feature_extractor feature_extractor(mesh_adaptor);
    int nb_borders = feature_extractor.get_nb_borders();
    int genus = feature_extractor.get_genus();

    // If mesh is a topological disk
    if (genus == 0 && nb_borders > 0)
    {
        // Pick the longest border
        seam = feature_extractor.get_longest_border();
    }
    else // if mesh is *not* a topological disk, create a virtual cut
    {
        const int CUT_LENGTH = 6;

        // Build consecutive halfedges array
        Polyhedron::Halfedge_handle seam_halfedges[CUT_LENGTH];
        seam_halfedges[0] = mesh.halfedges_begin();
        if (seam_halfedges[0] == NULL)
            return seam;                    // return empty list
        int i;
```

```
        for (i=1; i<CUT_LENGTH; i++)
        {
            seam_halfedges[i] =
    seam_halfedges[i-1]->next()->opposite()->next();
            if (seam_halfedges[i] == NULL)
                return seam;                // return empty list
        }

        // Convert halfedges array to two-ways vertices list
        for (i=0; i<CUT_LENGTH; i++)
            seam.push_back(seam_halfedges[i]->vertex());
        for (i=CUT_LENGTH-1; i>=0; i--)
            seam.push_back(seam_halfedges[i]->opposite()->vertex());
    }

    return seam;
}


//
    ----------------------------------------------------------------------------
// main()
//
    ----------------------------------------------------------------------------

int main(int argc, char * argv[])
{
    std::cerr << "PARAMETERIZATION" << std::endl;
    std::cerr << "  Floater parameterization" << std::endl;
    std::cerr << "  Circle border" << std::endl;
    std::cerr << "  OpenNL solver" << std::endl;
    std::cerr << "  Very simple cut if model is not a topological disk"
    << std::endl;

    //***************************************
    // decode parameters
    //***************************************

    if (argc-1 != 1)
    {
        std::cerr << "Usage: " << argv[0] << " input_file.off" <<
    std::endl;
        return(EXIT_FAILURE);
    }

    // File name is:
    const char* input_filename  = argv[1];

    //***************************************
    // Read the mesh
    //***************************************

    // Read the mesh
    std::ifstream stream(input_filename);
```

```
 Polyhedron mesh;
 stream >> mesh;
 if(!stream || !mesh.is_valid() || mesh.empty())
 {
      std::cerr << "Error: cannot read OFF file " << input_filename <<
std::endl;
      return EXIT_FAILURE;
 }

 //***************************************
 // Create Polyhedron adaptor
 //***************************************

 Parameterization_polyhedron_adaptor mesh_adaptor(mesh);

 //***************************************
 // Virtually cut mesh
 //***************************************

 // The parameterization methods support only meshes that
 // are topological disks => we need to compute a "cutting" of the
mesh
 // that makes it homeomorphic to a disk
 Seam seam = cut_mesh(mesh_adaptor);
 if (seam.empty())
 {
      std::cerr << "Input mesh not supported: the example cutting
algorithm is too simple to cut this shape" << std::endl;
      return EXIT_FAILURE;
 }

 // Create a second adaptor that virtually "cuts" the mesh following
the 'seam' path
 typedef
CGAL::Parameterization_mesh_patch_3<Parameterization_polyhedron_adaptor>
                                         Mesh_patch_polyhedron;
 Mesh_patch_polyhedron   mesh_patch(mesh_adaptor, seam.begin(),
seam.end());
 if (!mesh_patch.is_valid())
 {
      std::cerr << "Input mesh not supported: non manifold shape or
invalid cutting" << std::endl;
      return EXIT_FAILURE;
 }

 //***************************************
 // Floater Mean Value Coordinates parameterization
 //***************************************

 typedef CGAL::Parameterizer_traits_3<Mesh_patch_polyhedron>
                                         Parameterizer; // Type that
defines the error codes

 Parameterizer::Error_code err = CGAL::parameterize(mesh_patch);
```

```
    switch(err) {
    case Parameterizer::OK: // Success
        break;
    case Parameterizer::ERROR_EMPTY_MESH: // Input mesh not supported
    case Parameterizer::ERROR_NON_TRIANGULAR_MESH:
    case Parameterizer::ERROR_NO_TOPOLOGICAL_DISC:
    case Parameterizer::ERROR_BORDER_TOO_SHORT:
        std::cerr << "Input mesh not supported: " <<
   Parameterizer::get_error_message(err) << std::endl;
        return EXIT_FAILURE;
        break;
    default: // Error
        std::cerr << "Error: " << Parameterizer::get_error_message(err)
   << std::endl;
        return EXIT_FAILURE;
        break;
    };

    //***************************************
    // Output
    //***************************************

    // Raw output: dump (u,v) pairs
    Polyhedron::Vertex_const_iterator pVertex;
    for (pVertex = mesh.vertices_begin();
        pVertex != mesh.vertices_end();
        pVertex++)
    {
        // (u,v) pair is stored in any halfedge
        double u = mesh_adaptor.info(pVertex->halfedge())->uv().x();
        double v = mesh_adaptor.info(pVertex->halfedge())->uv().y();
        std::cout << "(u,v) = (" << u << "," << v << ")" << std::endl;
    }

    return EXIT_SUCCESS;
}
```

File: examples/Surface_mesh_parameterization/Mesh_cutting_parameterization.cpp

54.6 Output

Parameterization methods compute (u,v) fields for each vertex of the input mesh, with the seam vertices being virtually duplicated (thanks to *CGAL::Parameterization_mesh_patch_3<ParameterizationPatchableMesh_3>*). To support this duplication, *CGAL::Parameterization_polyhedron_adaptor_3<Polyhedron_3_>* stores the result in the (u,v) fields of the input mesh halfedges. A (u,v) pair is computed for each inner vertex (i.e. its halfedges share the same (u,v) pair), while a (u,v) pair is computed for each border halfedge. The user has to iterate over the mesh halfedges to get the result. Note that (u,v) fields do not exist in *CGAL::Polyhedron_3<Traits>*, thus the output traversal is specific to the way the (u,v) fields are implemented by the adaptor.

54.6.1 EPS Output Example

Complete_parameterization_example.cpp is a complete parameterization example which outputs the resulting parameterization to a EPS file. It gets the (u,v) fields computed by a parameterization method over a *CGAL::Polyhedron_3<Traits>* mesh with a *CGAL::Parameterization_polyhedron_adaptor_3<Polyhedron_3_>* adaptor:

```
#include <CGAL/basic.h> // include basic.h before testing #defines

#include <CGAL/Cartesian.h>
#include <CGAL/Polyhedron_3.h>
#include <CGAL/IO/Polyhedron_iostream.h>
#include <CGAL/Parameterization_polyhedron_adaptor_3.h>
#include <CGAL/parameterize.h>
#include <CGAL/Discrete_authalic_parameterizer_3.h>
#include <CGAL/Square_border_parameterizer_3.h>
#include <CGAL/Parameterization_mesh_patch_3.h>

#include <CGAL/Eigen_solver_traits.h>

#include <iostream>
#include <fstream>
#include <cstdlib>


//
   ----------------------------------------------------------------------------
// Private types
//
   ----------------------------------------------------------------------------

typedef CGAL::Cartesian<double>             Kernel;
typedef CGAL::Polyhedron_3<Kernel>          Polyhedron;

// Polyhedron adaptor
typedef CGAL::Parameterization_polyhedron_adaptor_3<Polyhedron>

   Parameterization_polyhedron_adaptor;

// Type describing a border or seam as a vertex list
typedef std::list<Parameterization_polyhedron_adaptor::Vertex_handle>
                                            Seam;


//
   ----------------------------------------------------------------------------
// Private functions
//
   ----------------------------------------------------------------------------

// If the mesh is a topological disk, extract its longest border,
// else compute a very simple cut to make it homeomorphic to a disk.
// Return the border of this region (empty on error)
//
```

```
// CAUTION: this cutting algorithm is very naive. Write your own!
static Seam cut_mesh(Parameterization_polyhedron_adaptor& mesh_adaptor)
{
    // Helper class to compute genus or extract borders
    typedef
   CGAL::Parameterization_mesh_feature_extractor<Parameterization_polyhedron_adaptor>
                                         Mesh_feature_extractor;

    Seam seam;               // returned list

    // Get reference to Polyhedron_3 mesh
    Polyhedron& mesh = mesh_adaptor.get_adapted_mesh();

    // Extract mesh borders and compute genus
    Mesh_feature_extractor feature_extractor(mesh_adaptor);
    int nb_borders = feature_extractor.get_nb_borders();
    int genus = feature_extractor.get_genus();

    // If mesh is a topological disk
    if (genus == 0 && nb_borders > 0)
    {
        // Pick the longest border
        seam = feature_extractor.get_longest_border();
    }
    else // if mesh is *not* a topological disk, create a virtual cut
    {
        const int CUT_LENGTH = 6;

        // Build consecutive halfedges array
        Polyhedron::Halfedge_handle seam_halfedges[CUT_LENGTH];
        seam_halfedges[0] = mesh.halfedges_begin();
        if (seam_halfedges[0] == NULL)
            return seam;                // return empty list
        int i;
        for (i=1; i<CUT_LENGTH; i++)
        {
            seam_halfedges[i] =
   seam_halfedges[i-1]->next()->opposite()->next();
            if (seam_halfedges[i] == NULL)
                return seam;            // return empty list
        }

        // Convert halfedges array to two-ways vertices list
        for (i=0; i<CUT_LENGTH; i++)
            seam.push_back(seam_halfedges[i]->vertex());
        for (i=CUT_LENGTH-1; i>=0; i--)
            seam.push_back(seam_halfedges[i]->opposite()->vertex());
    }

    return seam;
}

// Dump parameterized mesh to an eps file
static bool write_file_eps(const Parameterization_polyhedron_adaptor&
```

```
  mesh_adaptor,
                             const char *pFilename,
                             double scale = 500.0)
{
    const Polyhedron& mesh = mesh_adaptor.get_adapted_mesh();

    std::ofstream out(pFilename);
    if(!out)
        return false;
    CGAL::set_ascii_mode(out);

    // compute bounding box
    double xmin,xmax,ymin,ymax;
    xmin = ymin = xmax = ymax = 0;
    Polyhedron::Halfedge_const_iterator pHalfedge;
    for (pHalfedge = mesh.halfedges_begin();
         pHalfedge != mesh.halfedges_end();
         pHalfedge++)
    {
        double x1 = scale *
  mesh_adaptor.info(pHalfedge->prev())->uv().x();
        double y1 = scale *
  mesh_adaptor.info(pHalfedge->prev())->uv().y();
        double x2 = scale * mesh_adaptor.info(pHalfedge)->uv().x();
        double y2 = scale * mesh_adaptor.info(pHalfedge)->uv().y();
        xmin = (std::min)(xmin,x1);
        xmin = (std::min)(xmin,x2);
        xmax = (std::max)(xmax,x1);
        xmax = (std::max)(xmax,x2);
        ymax = (std::max)(ymax,y1);
        ymax = (std::max)(ymax,y2);
        ymin = (std::min)(ymin,y1);
        ymin = (std::min)(ymin,y2);
    }

    out << "%!PS-Adobe-2.0 EPSF-2.0" << std::endl;
    out << "%%BoundingBox: " << int(xmin+0.5) << " "
                                << int(ymin+0.5) << " "
                                << int(xmax+0.5) << " "
                                << int(ymax+0.5) << std::endl;
    out << "%%HiResBoundingBox: " << xmin << " "
                                    << ymin << " "
                                    << xmax << " "
                                    << ymax << std::endl;
    out << "%%EndComments" << std::endl;
    out << "gsave" << std::endl;
    out << "0.1 setlinewidth" << std::endl;

    // color macros
    out << std::endl;
    out << "% RGB color command - r g b C" << std::endl;
    out << "/C { setrgbcolor } bind def" << std::endl;
    out << "/white { 1 1 1 C } bind def" << std::endl;
    out << "/black { 0 0 0 C } bind def" << std::endl;
```

```
    // edge macro -> E
    out << std::endl;
    out << "% Black stroke - x1 y1 x2 y2 E" << std::endl;
    out << "/E {moveto lineto stroke} bind def" << std::endl;
    out << "black" << std::endl << std::endl;

    // for each halfedge
    for (pHalfedge = mesh.halfedges_begin();
         pHalfedge != mesh.halfedges_end();
         pHalfedge++)
    {
        double x1 = scale *
   mesh_adaptor.info(pHalfedge->prev())->uv().x();
        double y1 = scale *
   mesh_adaptor.info(pHalfedge->prev())->uv().y();
        double x2 = scale * mesh_adaptor.info(pHalfedge)->uv().x();
        double y2 = scale * mesh_adaptor.info(pHalfedge)->uv().y();
        out << x1 << " " << y1 << " " << x2 << " " << y2 << " E" <<
   std::endl;
    }

    /* Emit EPS trailer. */
    out << "grestore" << std::endl;
    out << std::endl;
    out << "showpage" << std::endl;

    return true;
}


//
    ---------------------------------------------------------------------
// main()
//
    ---------------------------------------------------------------------

int main(int argc, char * argv[])
{
    std::cerr << "PARAMETERIZATION" << std::endl;
    std::cerr << "  Discrete Authalic Parameterization" << std::endl;
    std::cerr << "  Square border" << std::endl;
    std::cerr << "  Eigen solver" << std::endl;
    std::cerr << "  Very simple cut if model is not a topological disk"
   << std::endl;
    std::cerr << "  Output: EPS" << std::endl;

    //***************************************
    // decode parameters
    //***************************************

    if (argc-1 != 2)
    {
        std::cerr << "Usage: " << argv[0] << " input_file.off
```

```
output_file.eps" << std::endl;
     return(EXIT_FAILURE);
 }

 // File names are:
 const char* input_filename  = argv[1];
 const char* output_filename = argv[2];

 //***************************************
 // Read the mesh
 //***************************************

 // Read the mesh
 std::ifstream stream(input_filename);
 Polyhedron mesh;
 stream >> mesh;
 if(!stream || !mesh.is_valid() || mesh.empty())
 {
     std::cerr << "Error: cannot read OFF file " << input_filename <<
std::endl;
     return EXIT_FAILURE;
 }

 //***************************************
 // Create Polyhedron adaptor
 //***************************************

 Parameterization_polyhedron_adaptor mesh_adaptor(mesh);

 //***************************************
 // Virtually cut mesh
 //***************************************

 // The parameterization methods support only meshes that
 // are topological disks => we need to compute a "cutting" of the
mesh
 // that makes it homeomorphic to a disk
 Seam seam = cut_mesh(mesh_adaptor);
 if (seam.empty())
 {
     std::cerr << "Input mesh not supported: the example cutting
algorithm is too simple to cut this shape" << std::endl;
     return EXIT_FAILURE;
 }

 // Create a second adaptor that virtually "cuts" the mesh following
the 'seam' path
 typedef
CGAL::Parameterization_mesh_patch_3<Parameterization_polyhedron_adaptor>
                                          Mesh_patch_polyhedron;
 Mesh_patch_polyhedron   mesh_patch(mesh_adaptor, seam.begin(),
seam.end());
 if (!mesh_patch.is_valid())
 {
```

```
     std::cerr << "Input mesh not supported: non manifold shape or
invalid cutting" << std::endl;
     return EXIT_FAILURE;
 }

 //***************************************
 // Discrete Authalic Parameterization (square border)
 // with Eigen solver
 //***************************************

 // Border parameterizer
 typedef
CGAL::Square_border_arc_length_parameterizer_3<Mesh_patch_polyhedron>

Border_parameterizer;
 // Eigen solver
 typedef CGAL::Eigen_solver_traits<>                Solver;

 // Discrete Authalic Parameterization (square border)
 // with Eigen solver
 typedef
CGAL::Discrete_authalic_parameterizer_3<Mesh_patch_polyhedron,
                                                Border_parameterizer,
                                                Solver>
Parameterizer;

 Parameterizer::Error_code err = CGAL::parameterize(mesh_patch,
Parameterizer());
 switch(err) {
 case Parameterizer::OK: // Success
     break;
 case Parameterizer::ERROR_EMPTY_MESH: // Input mesh not supported
 case Parameterizer::ERROR_NON_TRIANGULAR_MESH:
 case Parameterizer::ERROR_NO_TOPOLOGICAL_DISC:
 case Parameterizer::ERROR_BORDER_TOO_SHORT:
     std::cerr << "Input mesh not supported: " <<
Parameterizer::get_error_message(err) << std::endl;
     return EXIT_FAILURE;
     break;
 default: // Error
     std::cerr << "Error: " << Parameterizer::get_error_message(err)
<< std::endl;
     return EXIT_FAILURE;
     break;
 };

 //***************************************
 // Output
 //***************************************

 // Write Postscript file
 if ( ! write_file_eps(mesh_adaptor, output_filename) )
 {
     std::cerr << "Error: cannot write file " << output_filename <<
```

```
    std::endl;
        return EXIT_FAILURE;
    }

    return EXIT_SUCCESS;
}
```

File: examples/Surface_mesh_parameterization/Complete_parameterization_example.cpp

54.7 Complexity and Guarantees

54.7.1 Parameterization Methods and Guarantees

- Fixed boundaries
  - One-to-one mapping
    Tutte's theorem guarantees a one-to-one mapping provided that the weights are all positive and the border convex. It is the case for Tutte barycentric mapping and Floater mean value coordinates. It is not always the case for discrete conformal map (cotangents) and discrete authalic parameterization.
  - Non-singularity of the matrix
    Geshorgin's theorem guarantees the convergence of the solver if the matrix is diagonal dominant. This is the case with positive weights (Tutte barycentric mapping and Floater mean value coordinates).
- Free boundaries
  - One-to-one mapping
    No guarantee is provided by LSCM (both global overlaps and triangle flips can occur).
  - Non-singularity of the matrix
    For LSCM, the matrix of the system is the Gram matrix of a matrix with maximal rank, and is therefore non-singular (Gram theorem).

54.7.2 Precision

Two algorithms of this package construct the sparse linear system(s) using trigonometric functions, and are this incompatible with exact arithmetic:

- Floater mean value coordinates
- Circular border parameterization

On the other hand, linear solvers commonly use double precision floating point numbers.
OpenNL's BICGSTAB solver (accessible through the *OpenNL::DefaultLinearSolverTraits<COEFFTYPE, MATRIX, VECTOR, SOLVER>* interface) is the only solver supported by this package which computes exact results, when used with an exact arithmetic. This package is intended to be used mainly with a CGAL Cartesian kernel with doubles.

OpenNL's BICGSTAB Solver with an Exact Arithmetic

The BICGSTAB conjugate gradient is in disguise a direct solver. In a nutshell, it computes a vector basis orthogonal with respect to the matrix, and the coordinates of the solution in this vector basis. Each iteration computes one component of the basis and one coordinate, therefore the algorithm converges to the solution in n iterations, where n is the dimension of the matrix. More precisely, it is shown to converge in k iteration, where k is the number of distinct eigenvalues of the matrix.

Solvers with a Floating Point Arithmetic

OpenNL's BICGSTAB example:

When inexact numerical types are used (e.g. doubles), accumulated errors slow down convergence (in practice, it requires approximately $5k$ iterations to converge). The required number of iterations depends on the eigenvalues of the matrix, and these eigenvalues depend on the shape of the triangles. The optimum is when the triangles are equilateral (then the solver converges in less than 10 iterations). The worst case is obtained when the mesh has a large number of skinny triangles (near-singular Jacobian matrix of the triangle). In this case, the spectrum of the matrix is wide (many different eigenvalues), and the solver requires nearly $5n$ iterations to converge.

54.7.3 Algorithmic Complexity

In this package, we focus on piecewise linear mappings onto a planar domain. All surface parameterization methods are based on solving one (or two) sparse linear system(s). The algorithmic complexity is dominated by the resolution of the sparse linear system(s).

OpenNL's BICGSTAB example:

At each iteration, the operation of highest complexity is the product between the sparse-matrix and a vector. The sparse matrix has a fixed number of non-zero coefficients per row, therefore the matrix / vector product has $O(n)$ complexity. Since convergence is reached after k iterations, the complexity is $O(k.n)$ (where k is the number of distinct eigenvalues of the matrix). Therefore, best case complexity is $O(n)$ (equilateral triangles), and worst case complexity is $O(n^2)$ (skinny triangles).

54.8 Software Design

54.8.1 Global Function parameterize()

This package's entry point is:

```
// Compute a one-to-one mapping from a 3D triangle surface 'mesh' to a
// 2D circle, using Floater Mean Value Coordinates algorithm.
// A one-to-one mapping is guaranteed.
template <class ParameterizationMesh_3>
typename Parameterizer_traits_3<ParameterizationMesh_3>::Error_code
parameterize(ParameterizationMesh_3& mesh)  // 3D mesh, model of
    ParameterizationMesh_3 concept
{
```

```
    Mean_value_coordinates_parameterizer_3<ParameterizationMesh_3>
   parameterizer;
    return parameterizer.parameterize(mesh);
}

// Compute a one-to-one mapping from a 3D triangle surface 'mesh' to a
// simple 2D domain.
// One-to-one mapping may be guaranteed or not,
// depending on the chosen ParametizerTraits_3 algorithm.
template <class ParameterizationMesh_3, class ParameterizerTraits_3>
typename Parameterizer_traits_3<ParameterizationMesh_3>::Error_code
parameterize(ParameterizationMesh_3& mesh,          // 3D mesh, model of
    ParameterizationMesh_3
             ParameterizerTraits_3 parameterizer)   // Parameterization
    method for 'mesh'
{
    return parameterizer.parameterize(mesh);
}
```

You may notice that these global functions simply call the parameterize() method of a *ParameterizerTraits_3* object. The purpose of these global functions is:

- to be consistent with other CGAL algorithms that are also provided as global functions, e.g. *CGAL::convex_hull_2()*,
- to provide a default parameterization method (Floater Mean Value Coordinates), which wouldn't be possible with a direct call to an object's method.

You may also wonder why there is not just one *CGAL::parameterize()* function with a default *ParameterizerTraits_3* argument equal to *CGAL::Mean_value_coordinates_parameterizer_3< ParameterizationMesh_3>*. The reason is simply that this is not allowed by the C++ standard (see [C++98], paragraph 14.1/9).

54.8.2 No Common Parameterization Algorithm

ParameterizerTraits_3 models modify the behavior of the global function *CGAL::parameterize()* - hence the *Traits* in the name. On the other hand, *ParameterizerTraits_3* models do not modify the behavior of a common parameterization algorithm - as you might expect.

In this package, we focus on triangulated surfaces that are homeomorphic to a disk and on piecewise linear mappings onto planar domains. A consequence is that the skeleton of all parameterization methods of this package is the same:

- Allocate a sparse linear system $A.X = B$
- Parameterize the mesh border and initialize B
- Parameterize the inner points of the mesh and set A coefficients
- Solve the system

It is tempting to make the parameterization method a traits class that modifies the behavior of a common parameterization algorithm. On the other hand, there are several differences among methods:

- Fixed border methods need to parameterize all border vertices, while free border methods parameterize only two vertices.
- Some methods create symmetric definite positive systems, which may be solved more efficiently than general systems.
- Most parameterization methods use two #vertices x #vertices systems, where Least Squares Conformal Maps uses one (2 * #triangles) x #vertices system.
- Most parameterization methods invert the A matrix, when Least Squares Conformal Maps solves the system in the least squares sense.

Therefore, the software design chosen is:

- Each *ParameterizerTraits_3* model implements its own version of the parameterization algorithm as a parameterize() method.
- Each *ParameterizerTraits_3* model has template arguments defining the border parameterization and sparse linear solver to use, with default values adapted to the method.
- Code factorization is achieved using a class hierarchy and (few) virtual methods.

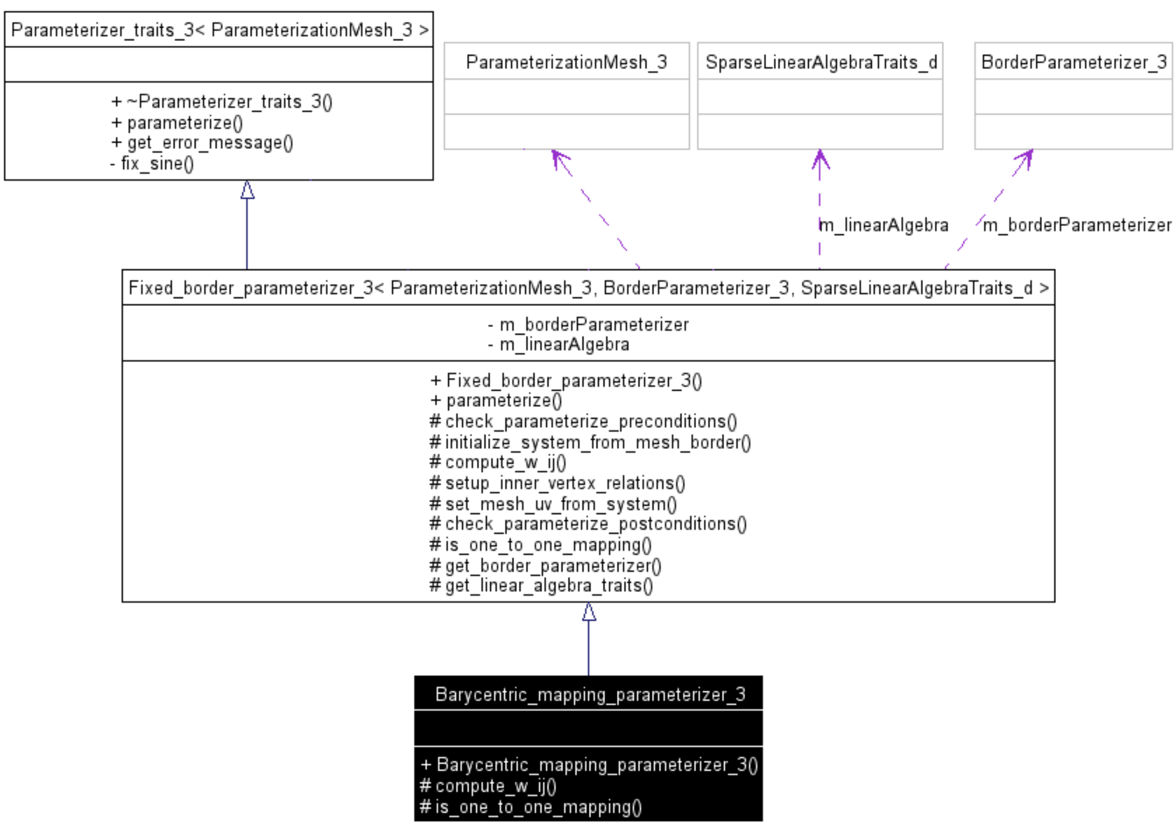


Figure 54.10: A parameterizer UML class diagram (main types and methods only)

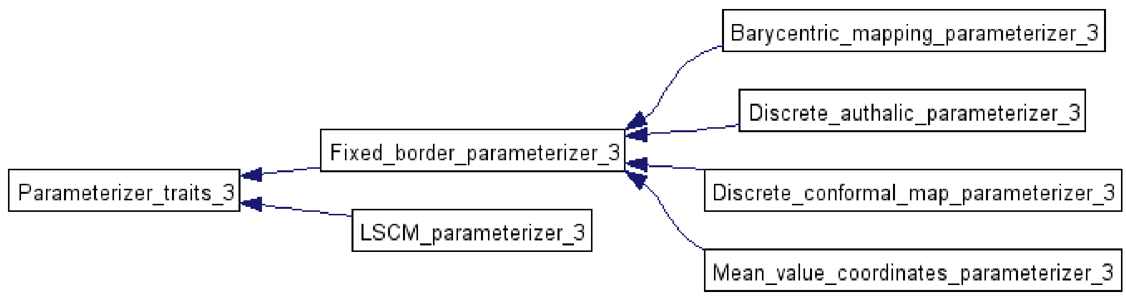


Figure 54.11: Surface parameterizer classes hierarchy

Note: *CGAL::Parameterizer_traits_3<ParameterizationMesh_3>* is the (pure virtual) superclass of all surface parameterization classes.

54.8.3 Fixed_border_parameterizer_3 Class

Linear fixed border parameterization algorithms are very close. They mainly differ by the energy that they try to minimize, i.e. by the value of the w_{ij} coefficient of the A matrix, for v_i and v_j neighbor vertices of the mesh [FH05]. One consequence is that most of the code of the fixed border methods is factorized in the *CGAL::Fixed_border_parameterizer_3<ParameterizationMesh_3, BorderParameterizer_3, SparseLinearAlgebraTraits_d>* class.

Subclasses:

- must provide *BorderParameterizer_3* and *SparseLinearAlgebraTraits_d* default template parameters that make sense,
- must implement *compute_w_ij*() to compute w_{ij} = (i, j) coefficient of matrix A for v_j neighbor vertex of v_i,
- may implement an optimized version of *is_one_to_one_mapping*().

See *CGAL::Barycentric_mapping_parameterizer_3<ParameterizationMesh_3, BorderParameterizer_3, SparseLinearAlgebraTraits_d>* class as an example.

54.8.4 Border Parameterizations

Border Parameterizations are models of the *BorderParameterizer_3* concept. To simplify the implementation, *BorderParameterizer_3* models know only the *ParameterizationMesh_3* mesh class. They do not know the parameterization algorithm or the sparse linear solver used.

54.8.5 ParameterizationMesh_3 and ParameterizationPatchableMesh_3 Concepts

All parameterization methods are templated by the kind of mesh they are applied on. The mesh type must be a model of *ParameterizationMesh_3*.

The purpose of such a model is to:

1. Support several kind of meshes.
2. Hide the implementation of extra fields specific to the parameterization domain (*index*, *u*, *v*, *is_parameterized*).
3. Handle in the mesh type the complexity of *virtually* cutting a mesh to make it homeomorphic to a disk (instead of duplicating this code in each parameterization method).

Two options are possible for 1) and 2):

- Pass to all classes and methods a mesh pointer, a traits class to manipulate it, and accessors to the extra field arrays. This is the choice of the Boost Graph Library with *boost::graph_traits<>* and the property maps.
- Pass to all classes and methods an object that points to the actual mesh and knows how to access to its fields. This is the Adaptor concept [GHJV95].

The current design of this package uses the second option, which is simpler. Of course, we may decide at some point to switch to the first one to reach a deeper integration of CGAL with Boost.

Point 3) is solved by class *CGAL::Parameterization_mesh_patch_3<ParameterizationPatchableMesh_3>*, which takes care of *virtually* cutting a patch in a *ParameterizationPatchableMesh_3* mesh, to make it appear as a topological disk with a *ParameterizationMesh_3* interface. *ParameterizationPatchableMesh_3* inherits from concept *ParameterizationMesh_3* and adds the ability to support patches and virtual seams.

This mainly means that:

- vertices can be tagged as inside or outside the patch to parameterize,
- the fields specific to parameterizations (*index*, *u*, *v*, *is_parameterized*) can be set *per corner* (which is a more general way of saying *per half-edge*).

54.8.6 SparseLinearAlgebraTraits_d Concept

This package solves sparse linear systems using solvers which are models of *SparseLinearAlgebraTraits_d*.

SparseLinearAlgebraTraits_d is a sub-concept of the *LinearAlgebraTraits_d* concept in *Kernel_d*. The goal is to adapt easily code written for dense matrices to sparse ones, and vice-versa.

54.8.7 Cutting a Mesh

In this package, we focus on triangulated surfaces that are homeomorphic to a disk.

Computing a cutting path that transforms a closed mesh of arbitrary genus into a topological disk is a research topic on its own. This package does not intend to cover this topic at the moment.

54.9 Extending the Package and Reusing Code

54.9.1 Reusing Mesh Adaptors

ParameterizationMesh_3 defines a concept to access to a general polyhedral mesh. It is optimized for the *Surface_mesh_parameterization* package only in the sense that it defines the accessors to fields specific to the parameterization domain (*index*, *u*, *v*, *is_parameterized*).

It may be easily generalized.

54.9.2 Reusing Sparse Linear Algebra

The *SparseLinearAlgebraTraits_d* concept and the traits classes for EIGEN and OpenNL are independent of the rest of the *Surface_mesh_parameterization* package, and may be reused by CGAL developers for other purposes.

54.9.3 Adding New Parameterization Methods

Implementing a new fixed border linear parameterization is easy. Most of the code of the fixed border methods is factorized in the *CGAL::Fixed_border_parameterizer_3<ParameterizationMesh_3, BorderParameterizer_3, SparseLinearAlgebraTraits_d>* class. Subclasses must mainly implement a *compute_w_ij*() method which computes each $w_{ij} = (i, j)$ coefficient of the matrix A for v_j neighboring vertices of v_i.

Although implementing a new free border linear parameterization method is more challenging, the Least Squares Conformal Maps parameterization method provides a good starting point.

Implementing *non* linear parameterizations is a natural extension to this package, although only the mesh adaptors can be reused.

54.9.4 Adding New Border Parameterization Methods

Implementing a new border parameterization method is easy. Square, circular and two-points border parameterizations are good starting points.

54.9.5 Mesh Cutting

Obviously, this package would benefit of having robust algorithms which transform arbitrary meshes into topological disks.

Planar Parameterization of Triangulated Surface Meshes
Reference Manual

Laurent Saboret, Pierre Alliez and Bruno Lévy

Parameterizing a surface amounts to finding a one-to-one mapping from a suitable domain to the surface. A good mapping is the one which minimizes either angle or area distortions in some sense. In this package, we focus on triangulated surfaces that are homeomorphic to a disk and on piecewise linear mappings into a planar domain.

54.10 Classified Reference Pages

Main Function

Concepts

Surface Parameterization Methods

This CGAL package implements some of the state-of-the-art parameterization methods:

- Fixed border:
  - Tutte Barycentric Mapping [Tut63]. One-to-one mapping is guaranteed for convex border.
  - Floater Mean Value Coordinates [Flo03a]. One-to-one mapping is guaranteed for convex border.

– Discrete Conformal Map [EDD+95]. Conditionally guaranteed if all weights are positive and border is convex.
– Discrete Authalic parameterization [DMA02]. Conditionally guaranteed if all weights are positive and border is convex.

- Free border:
  – Least Squares Conformal Maps [LPRM02].

Border Parameterization Methods

Border parameterization methods define a set of constraints (a constraint specifies two (u,v) coordinates for each instance of a vertex along the border).

This package implements all common border parameterization methods:

- For fixed border methods:
  – the user can select a border parameterization among two common methods: uniform or arc-length parameterizations.
  – one convex shape specified by:
    ∗ one shape among a set of standard ones (circle, square).
- For free border methods: at least two constraints (the pinned vertices).

Mesh

The general definition of input meshes handled *directly* by *CGAL::parameterize()* is:

- Model of *ParameterizationMesh_3*.
- Triangulated.
- 2-manifold.
- Oriented.
- Homeomorphic to a disc (may have holes).

This package provides a model of the *ParameterizationMesh_3* concept to access *CGAL::Polyhedron_3<Traits>* :
CGAL::Parameterization_polyhedron_adaptor_3<Polyhedron_3_>

The meshes supported *indirectly* by the package can be of any genus and have any number of connected components. If it is not a topological disc, the input mesh has to come with a description of a cutting path (an oriented list of vertices) which is the border of a topological disc. If no cutting path is given as input, we assume that the surface border is the longest border already in the input mesh (the other borders will be considered as holes).

The *CGAL::Parameterization_mesh_patch_3<ParameterizationPatchableMesh_3>* class is responsible for *virtually* cutting a patch in a *ParameterizationPatchableMesh_3* mesh. The resulting patch is a topological disk (if the input cutting path is correct) and provides a *ParameterizationMesh_3* interface. It can be used as parameter of *CGAL::parameterize()*.

Note that this way the user is responsible for cutting a closed mesh of arbitrary genus (even a topological disc with an intricate seam cut), as long as this condition is fulfilled.

The package provides an interface with *CGAL::Polyhedron_3<Traits>*:

Output

A (u,v) pair is computed for each inner vertex (i.e. its halfedges share the same (u,v) pair), while a (u,v) pair is computed for each border halfedge. The user must iterate over the mesh halfedges to get the result.

Sparse Linear Algebra

Since parameterizing meshes requires efficient representation of sparse matrices and efficient iterative or direct linear solvers, we provide an interface to several sparse linear solvers:

- EIGEN 3.1 (or greater) is the library recommended by CGALfor solving sparse systems.
- OpenNL (authored by Bruno Lévy) is shipped with CGALand is the default solver.

OpenNL::DefaultLinearSolverTraits<COEFFTYPE, MATRIX, VECTOR, SOLVER> in OpenNL package
OpenNL::SymmetricLinearSolverTraits<COEFFTYPE, MATRIX, VECTOR, SOLVER> in OpenNL package

Helper Classes

Checks and Assertions

The package performs the next checks:

For *fixed* border parameterizations:

- Preconditions:
  - check that the border is mapped onto a convex polygon.
  - check that the input mesh is triangular.
  - check that the input mesh is a surface with one connected component.
- Postconditions:
  - check one-to-one mapping.

For *free* border parameterizations:

- Preconditions:
  - check that the input mesh is triangular.
  - check that the input mesh is a surface with one connected component.
- Postconditions:
  - check one-to-one mapping.

Assertions are optional checks. The assertion flags for the package use *SURFACE_MESH_PARAMETERIZATION* in their names (*e.g.*, *CGAL_SURFACE_MESH_PARAMETERIZATION_NO_ASSERTIONS*).

54.11 Alphabetical List of Reference Pages

Class

CGAL::Barycentric_mapping_parameterizer_3<ParameterizationMesh_3, BorderParameterizer_3, SparseLinearAlgebraTraits_d>

Definition

The class *Barycentric_mapping_parameterizer_3* implements Tutte Barycentric Mapping algorithm [Tut63]. This algorithm is also called *Tutte Uniform Weights* by other authors.

One-to-one mapping is guaranteed if the surface's border is mapped to a convex polygon.

This class is a Strategy [GHJV95] called by the main parameterization algorithm *Fixed_border_parameterizer_3::parameterize*(). It provides default *BorderParameterizer_3* and *SparseLinearAlgebraTraits_d* template parameters that make sense. It implements *compute_w_ij*() to compute *w_ij* = (i, j) coefficient of matrix A for j neighbor vertex of i based on Tutte Barycentric Mapping method. It implements an optimized version of *is_one_to_one_mapping*().

#include <CGAL/Barycentric_mapping_parameterizer_3.h>

Inherits From

Fixed_border_parameterizer_3<ParameterizationMesh_3, BorderParameterizer_3, SparseLinearAlgebraTraits_d>

Is Model for the Concepts

Model of the *ParameterizerTraits_3* concept.

Design Pattern

Barycentric_mapping_parameterizer_3 class is a Strategy [GHJV95]: it implements a strategy of surface parameterization for models of *ParameterizationMesh_3*.

Parameters

The full template declaration is:

template<class *ParameterizationMesh_3*, class *BorderParameterizer_3* = *Circular_border_arc_length_parameterizer_3<ParameterizationMesh_3>*, class *SparseLinearAlgebraTraits_d* = *OpenNL::DefaultLinearSolverTraits<typename ParameterizationMesh_3::NT>*>
class *Barycentric_mapping_parameterizer_3*;

Parameters:
ParameterizationMesh_3: 3D surface mesh. *BorderParameterizer_3*: Strategy to parameterize the surface border. *SparseLinearAlgebraTraits_d*: Traits class to solve a sparse linear system. Note: the system is **not** symmetric because *Fixed_border_parameterizer_3* does not remove (yet) border vertices from the system.

Creation

Barycentric_mapping_parameterizer_3<ParameterizationMesh_3, BorderParameterizer_3, SparseLinearAlgebraTraits_d> param(Border_param border_param = Border_param(),

S

LA sparse_la = Sparse_LA())

Constructor.
Parameters:
border_param: Object that maps the surface's border to 2D space. *sparse_la*: Traits object to access a sparse linear system.

Operations

virtual NT *param.compute_w_ij(Adaptor, Vertex_const_handle, Vertex_around_vertex_const_circulator) [protected, virtual]*

Compute *w_ij* = (i, j) coefficient of matrix A for j neighbor vertex of i.
Tutte Barycentric Mapping algorithm is the most simple one: *w_ij* = 1 for j neighbor vertex of i.

virtual bool *param.is_one_to_one_mapping(Adaptor, Matrix, Vector, Vector) [protected, virtual]*

Check if 3D -> 2D mapping is one-to-one.
Theorem: one-to-one mapping is guaranteed if all *w_ij* coefficients are > 0 (for j vertex neighbor of i) and if the surface border is mapped onto a 2D convex polygon. All *w_ij* coefficients = 1 (for j vertex neighbor of i), thus mapping is guaranteed if the surface border is mapped onto a 2D convex polygon.

See Also

CGAL::Parameterizer_traits_3<ParameterizationMesh_3> . . . . page 3393
CGAL::Fixed_border_parameterizer_3<ParameterizationMesh_3, BorderParameterizer_3, SparseLinearAlgebraTraits_d> . . . . page 3362
CGAL::Discrete_authalic_parameterizer_3<ParameterizationMesh_3, BorderParameterizer_3, SparseLinearAlgebraTraits_d> . . . . page 3358
CGAL::Discrete_conformal_map_parameterizer_3<ParameterizationMesh_3, BorderParameterizer_3, SparseLinearAlgebraTraits_d> . . . . page 3360
CGAL::LSCM_parameterizer_3<ParameterizationMesh_3, BorderParameterizer_3, SparseLinearAlgebraTraits_d> . . . . page 3365
CGAL::Mean_value_coordinates_parameterizer_3<ParameterizationMesh_3, BorderParameterizer_3, SparseLinearAlgebraTraits_d> . . . . page 3369

Concept

BorderParameterizer_3

Definition

BorderParameterizer_3 is a concept of class that parameterizes a given type of mesh, *Adaptor*, which is a model of the *ParameterizationMesh_3* concept.

Implementation note: To simplify the implementation, *BorderParameterizer_3* models know only the *ParameterizationMesh_3* class. They do not know the parameterization algorithm requirements or the kind of sparse linear system used.

Design Pattern

BorderParameterizer_3 models are Strategies [GHJV95]: they implement a strategy of border parameterization for models of *ParameterizationMesh_3*.

Types

BorderParameterizer_3:: Adaptor	Export *ParameterizationMesh_3* template parameter.
BorderParameterizer_3:: Error_code	The various errors detected by this package.

Creation

Construction and destruction are undefined.

Operations

Error_code	*bp.parameterize_border(Adaptor& mesh)*	
		Assign to mesh's border vertices a 2D position (i.e. a (u, v) pair) on border's shape. Mark them as *parameterized*. Return false on error.
bool	*bp.is_border_convex()*	
		Indicate if border's shape is convex.

Has Models

CGAL::Circular_border_arc_length_parameterizer_3<ParameterizationMesh_3> . . . . page 3353
CGAL::Circular_border_uniform_parameterizer_3<ParameterizationMesh_3> . . . . page 3357
CGAL::Square_border_arc_length_parameterizer_3<ParameterizationMesh_3> . . . . page 3401
CGAL::Square_border_uniform_parameterizer_3<ParameterizationMesh_3> . . . . page 3405
CGAL::Two_vertices_parameterizer_3<ParameterizationMesh_3> . . . . page 3412

See Also

ParameterizerTraits_3 . . . . page 3396
ParameterizationMesh_3 . . . . page 3371

CGAL::Circular_border_arc_length_parameterizer_3< ParameterizationMesh_3>

Class

Definition

Circular_border_arc_length_parameterizer_3 is the default border parameterizer for fixed border parameterization methods.

This class parameterizes the border of a 3D surface onto a circle, with an arc-length parameterization: (u, v) values are proportional to the length of border edges. *Circular_border_parameterizer_3* implements most of the border parameterization algorithm. This class implements only *compute_edge_length*() to compute a segment's length.

#include <CGAL/Circular_border_parameterizer_3.h>

Inherits From

Circular_border_parameterizer_3<ParameterizationMesh_3>

Is Model for the Concepts

Model of the *BorderParameterizer_3* concept.

Design Pattern

BorderParameterizer_3 models are Strategies [GHJV95]: they implement a strategy of border parameterization for models of *ParameterizationMesh_3*

Parameters

The full template declaration is:

template<class *ParameterizationMesh_3*>
class *Circular_border_arc_length_parameterizer_3*;

Creation

Operations

virtual double *bp.compute_edge_length(Adaptor mesh,*
Vertex_const_handle source,
Vertex_const_handle target) [protected, virtual]

Compute the length of an edge.
Arc-length border parameterization: (u, v) values are proportional to the length of border edges.

See Also

CGAL::Circular_border_uniform_parameterizer_3<ParameterizationMesh_3> . . . . page 3357
CGAL::Square_border_arc_length_parameterizer_3<ParameterizationMesh_3> . . . . page 3401
CGAL::Square_border_uniform_parameterizer_3<ParameterizationMesh_3> . . . . page 3405
CGAL::Two_vertices_parameterizer_3<ParameterizationMesh_3> . . . . page 3412

Example

See *Eigen_parameterization.cpp* example.

CGAL::Circular_border_parameterizer_3<ParameterizationMesh_3>

Class

Definition

This is the base class of strategies that parameterize the border of a 3D surface onto a circle. *Circular_border_parameterizer_3* is a pure virtual class, thus cannot be instantiated. It implements most of the algorithm. Subclasses just have to implement *compute_edge_length*() to compute a segment's length.

Implementation note: To simplify the implementation, *BorderParameterizer_3* models know only the *ParameterizationMesh_3* class. They do not know the parameterization algorithm requirements or the kind of sparse linear system used.

#include <CGAL/Circular_border_parameterizer_3.h>

Is Model for the Concepts

Model of the *BorderParameterizer_3* concept (although you cannot instantiate this class).

Design Pattern

BorderParameterizer_3 models are Strategies [GHJV95]: they implement a strategy of border parameterization for models of *ParameterizationMesh_3*.

Parameters

The full template declaration is:

template<class *ParameterizationMesh_3*>
class *Circular_border_parameterizer_3*;

Types

Circular_border_parameterizer_3<ParameterizationMesh_3>:: Adaptor

Export *ParameterizationMesh_3* template parameter.

Creation

Operations

Parameterizer_traits_3<Adaptor>::Error_code

bp.parameterize_border(Adaptor& mesh)

Assign to mesh's border vertices a 2D position (i.e. a (u, v) pair) on border's shape. Mark them as *parameterized*.

bool	*bp.is_border_convex()*	
		Indicate if border's shape is convex.
virtual double	*bp.compute_edge_length(Adaptor mesh, Vertex_const_handle source, Vertex_const_handle target) [protected, pure virtual]*	
		Compute the length of an edge.

See Also

CGAL::Circular_border_arc_length_parameterizer_3<ParameterizationMesh_3> . . . . page 3353
CGAL::Circular_border_uniform_parameterizer_3<ParameterizationMesh_3> . . . . page 3357

CGAL::Circular_border_uniform_parameterizer_3< ParameterizationMesh_3>

Class

Definition

This class parameterizes the border of a 3D surface onto a circle in a uniform manner: points are equally spaced. *Circular_border_parameterizer_3* implements most of the border parameterization algorithm. This class implements only *compute_edge_length*() to compute a segment's length.

#include <CGAL/Circular_border_parameterizer_3.h>

Inherits From

Circular_border_parameterizer_3<ParameterizationMesh_3>

Is Model for the Concepts

Model of the *BorderParameterizer_3* concept.

Design Pattern

BorderParameterizer_3 models are Strategies [GHJV95]: they implement a strategy of border parameterization for models of *ParameterizationMesh_3*

Parameters

The full template declaration is:

template<class *ParameterizationMesh_3*>
class *Circular_border_uniform_parameterizer_3*;

Creation

Operations

virtual double	*bp.compute_edge_length(Adaptor, Vertex_const_handle, Vertex_const_handle) [protected, virtual]*
	Compute the length of an edge. Uniform border parameterization: points are equally spaced.

See Also

Class

CGAL::Discrete_authalic_parameterizer_3<ParameterizationMesh_3, BorderParameterizer_3, SparseLinearAlgebraTraits_d>

Definition

The class *Discrete_authalic_parameterizer_3* implements the Discrete Authalic Parameterization algorithm [DMA02]. This method is sometimes called *DAP* or just *Authalic parameterization*.

DAP is a weak area-preserving parameterization. It is a compromise between area-preserving and angle-preserving.

One-to-one mapping is guaranteed if surface's border is mapped onto a convex polygon.

This class is a Strategy [GHJV95] called by the main parameterization algorithm *Fixed_border_parameterizer_3::parameterize*(). *Discrete_authalic_parameterizer_3*: It provides default *BorderParameterizer_3* and *SparseLinearAlgebraTraits_d* template parameters that make sense. It implements *compute_w_ij*() to compute *w_ij* = (i, j) coefficient of matrix A for j neighbor vertex of i based on Discrete Authalic Parameterization algorithm.

#include <CGAL/Discrete_authalic_parameterizer_3.h>

Inherits From

Fixed_border_parameterizer_3<ParameterizationMesh_3, BorderParameterizer_3, SparseLinearAlgebraTraits_d>

Is Model for the Concepts

Model of the *ParameterizerTraits_3* concept.

Design Pattern

Discrete_authalic_parameterizer_3 class is a Strategy [GHJV95]: it implements a strategy of surface parameterization for models of *ParameterizationMesh_3*.

Parameters

The full template declaration is:

template<class *ParameterizationMesh_3*, class *BorderParameterizer_3* = *Circular_border_arc_length_parameterizer_3<ParameterizationMesh_3>*, class *SparseLinearAlgebraTraits_d* = *OpenNL::DefaultLinearSolverTraits<typename ParameterizationMesh_3::NT>*>
class *Discrete_authalic_parameterizer_3*;

Creation

Discrete_authalic_parameterizer_3<ParameterizationMesh_3, BorderParameterizer_3, SparseLinearAlgebraTraits_d> param(Border_param border_param = Border_param(),

Spars

LA sparse_la = Sparse_LA())

Constructor.
Parameters:
border_param: Object that maps the surface's border to 2D space. *sparse_la*: Traits object to access a sparse linear system.

Operations

virtual NT *param.compute_w_ij(Adaptor mesh, Vertex_const_handle main_vertex_v_i, Vertex_around_vertex_const_circulator neighbor_vertex_v_j)*

[protected, virtual]

Compute *w_ij* = (i, j) coefficient of matrix A for j neighbor vertex of i.

See Also

CGAL::Parameterizer_traits_3<ParameterizationMesh_3> . . . . page 3393
CGAL::Fixed_border_parameterizer_3<ParameterizationMesh_3, BorderParameterizer_3, SparseLinearAlgebraTraits_d> . . . . page 3362
CGAL::Barycentric_mapping_parameterizer_3<ParameterizationMesh_3, BorderParameterizer_3, SparseLinearAlgebraTraits_d> . . . . page 3350
CGAL::Discrete_conformal_map_parameterizer_3<ParameterizationMesh_3, BorderParameterizer_3, SparseLinearAlgebraTraits_d> . . . . page 3360
CGAL::LSCM_parameterizer_3<ParameterizationMesh_3, BorderParameterizer_3, SparseLinearAlgebraTraits_d> . . . . page 3365
CGAL::Mean_value_coordinates_parameterizer_3<ParameterizationMesh_3, BorderParameterizer_3, SparseLinearAlgebraTraits_d> . . . . page 3369

Example

See *Authalic_parameterization.cpp* example.

Class

CGAL::Discrete_conformal_map_parameterizer_3< ParameterizationMesh_3, BorderParameterizer_3, SparseLinearAlgebraTraits_d>

Definition

The class *Discrete_conformal_map_parameterizer_3* implements the Discrete Conformal Map (DCM) parameterization [EDD+95]. This algorithm is also called *Discrete Conformal Parameterization (DCP)*, *Discrete Harmonic Map* or *Fixed Conformal Parameterization* by other authors.

This is a conformal parameterization, i.e. it attempts to preserve angles.

One-to-one mapping is guaranteed if surface's border is mapped onto a convex polygon.

This class is a Strategy [GHJV95] called by the main parameterization algorithm *Fixed_border_parameterizer_3::parameterize*(). It provides default *BorderParameterizer_3* and *SparseLinearAlgebraTraits_d* template parameters that make sense. It implements *compute_w_ij*() to compute *w_ij* = (i, j) coefficient of matrix A for j neighbor vertex of i based on Discrete Conformal Map method.

#include <CGAL/Discrete_conformal_map_parameterizer_3.h>

Inherits From

Fixed_border_parameterizer_3<ParameterizationMesh_3, BorderParameterizer_3, SparseLinearAlgebraTraits_d>

Is Model for the Concepts

Model of the *ParameterizerTraits_3* concept.

Design Pattern

Discrete_conformal_map_parameterizer_3 class is a Strategy [GHJV95]: it implements a strategy of surface parameterization for models of *ParameterizationMesh_3*.

Parameters

The full template declaration is:

template<class *ParameterizationMesh_3*, class *BorderParameterizer_3* = *Circular_border_arc_length_parameterizer_3<ParameterizationMesh_3>*, class *SparseLinearAlgebraTraits_d* = *OpenNL::DefaultLinearSolverTraits<typename ParameterizationMesh_3::NT>*>
class *Discrete_conformal_map_parameterizer_3*;

Parameters:
ParameterizationMesh_3: 3D surface mesh. *BorderParameterizer_3*: Strategy to parameterize the surface border. *SparseLinearAlgebraTraits_d*: Traits class to solve a sparse linear system. Note: the system is **not** symmetric because *Fixed_border_parameterizer_3* does not remove (yet) border vertices from the system.

Creation

Discrete_conformal_map_parameterizer_3<ParameterizationMesh_3, BorderParameterizer_3, SparseLinearAlgebraTraits_d> param(Border_param border_param = Border_param(),

LA sparse_la = Sparse_LA())

Constructor.
Parameters:
border_param: Object that maps the surface's border to 2D space. *sparse_la*: Traits object to access a sparse linear system.

Operations

virtual NT *param.compute_w_ij(Adaptor mesh, Vertex_const_handle main_vertex_v_i, Vertex_around_vertex_const_circulator neighbor_vertex_v_j)*

[protected, virtual]

Compute *w_ij* = (i, j) coefficient of matrix A for j neighbor vertex of i.

See Also

CGAL::Parameterizer_traits_3<ParameterizationMesh_3> . . . . . . page 3393
CGAL::Fixed_border_parameterizer_3<ParameterizationMesh_3, BorderParameterizer_3, SparseLinearAlgebraTraits_d> . . . . . . page 3362
CGAL::Barycentric_mapping_parameterizer_3<ParameterizationMesh_3, BorderParameterizer_3, SparseLinearAlgebraTraits_d> . . . . . . page 3350
CGAL::Discrete_authalic_parameterizer_3<ParameterizationMesh_3, BorderParameterizer_3, SparseLinearAlgebraTraits_d> . . . . . . page 3358
CGAL::LSCM_parameterizer_3<ParameterizationMesh_3, BorderParameterizer_3, SparseLinearAlgebraTraits_d> . . . . . . page 3365
CGAL::Mean_value_coordinates_parameterizer_3<ParameterizationMesh_3, BorderParameterizer_3, SparseLinearAlgebraTraits_d> . . . . . . page 3369

Class

CGAL::Fixed_border_parameterizer_3<ParameterizationMesh_3, BorderParameterizer_3, SparseLinearAlgebraTraits_d>

Definition

The class *Fixed_border_parameterizer_3* is the base class of fixed border parameterization methods (Tutte, Floater, ...).

One-to-one mapping is guaranteed if surface's border is mapped onto a convex polygon.

This class is a pure virtual class, thus cannot be instantiated. Anyway, it implements most of the parameterization algorithm parameterize(). Subclasses are Strategies [GHJV95] that modify the behavior of this algorithm: They provide *BorderParameterizer_3* and *SparseLinearAlgebraTraits_d* template parameters that make sense. They implement *compute_w_ij*() to compute *w_ij* = (i, j) coefficient of matrix A for j neighbor vertex of i. They may implement an optimized version of *is_one_to_one_mapping*().

#include <CGAL/Fixed_border_parameterizer_3.h>

Inherits From

Parameterizer_traits_3<ParameterizationMesh_3>

Is Model for the Concepts

Model of the *ParameterizerTraits_3* concept (although you cannot instantiate this class).

Design Pattern

Fixed_border_parameterizer_3 class is a Strategy [GHJV95]: it implements (part of) a strategy of surface parameterization for models of *ParameterizationMesh_3*.

Parameters

The full template declaration is:

template<class *ParameterizationMesh_3*, class *BorderParameterizer_3* = *Circular_border_arc_length_parameterizer_3<ParameterizationMesh_3>*, class *SparseLinearAlgebraTraits_d* = *OpenNL::DefaultLinearSolverTraits<typename ParameterizationMesh_3::NT>*>
class *Fixed_border_parameterizer_3*;

Types

Fixed_border_parameterizer_3<ParameterizationMesh_3, BorderParameterizer_3, SparseLinearAlgebraTraits_d>:: Border_param

Export *BorderParameterizer_3* template parameter.

Fixed_border_parameterizer_3<ParameterizationMesh_3, BorderParameterizer_3, SparseLinearAlgebraTraits_d>:: Sparse_LA

Export *SparseLinearAlgebraTraits_d* template parameter.

Creation

Fixed_border_parameterizer_3<ParameterizationMesh_3, BorderParameterizer_3, SparseLinearAlgebraTraits_d> param(Border_param border_param = Border_param(), Sparse_LA sparse_la = Sparse_LA())

Constructor.
Parameters:
border_param: Object that maps the surface's border to 2D space. *sparse_la*: Traits object to access a sparse linear system.

Operations

Error_code *param.parameterize(Adaptor& mesh)*

Compute a one-to-one mapping from a triangular 3D surface *mesh* to a piece of the 2D space. The mapping is linear by pieces (linear in each triangle). The result is the (u, v) pair image of each vertex of the 3D surface.
Preconditions:
mesh must be a surface with one connected component. *mesh* must be a triangular mesh. The mesh border must be mapped onto a convex polygon.

Error_code *param.check_parameterize_preconditions(Adaptor& mesh) [protected, virtual]*

Check parameterize() preconditions: *mesh* must be a surface with one connected component. *mesh* must be a triangular mesh. The mesh border must be mapped onto a convex polygon.

void *param.initialize_system_from_mesh_border(Matrix& A, Vector& Bu, Vector& Bv, Adaptor mesh)*

Initialize A, Bu and Bv after border parameterization. Fill the border vertices' lines in both linear systems: *u = constant* and *v = constant*.
Preconditions:
Vertices must be indexed. A, Bu and Bv must be allocated. Border vertices must be parameterized.

virtual NT *param.compute_w_ij(Adaptor mesh, Vertex_const_handle main_vertex_v_i, Vertex_around_vertex_const_circulator neighbor_vertex_v_j) [protected, pure virtual]*

Compute *w_ij* = (i, j) coefficient of matrix A for j neighbor vertex of i. Implementation note: Subclasses must at least implement *compute_w_ij*().

Error_code *param.setup_inner_vertex_relations(Matrix& A, Vector& Bu, Vector& Bv, Adaptor mesh, Vertex_const_handle vertex) [protected, virtual]*

Compute the line i of matrix A for i inner vertex: call *compute_w_ij*() to compute the A coefficient *w_ij* for each neighbor *v_j*. compute *w_ii* = - sum of *w_ijs*.
Preconditions:
Vertices must be indexed. Vertex i musn't be already parameterized. Line i of A must contain only zeros.

void *param.set_mesh_uv_from_system(Adaptor& mesh, Vector Xu, Vector Xv)*

Copy Xu and Xv coordinates into the (u, v) pair of each surface vertex.

Error_code *param.check_parameterize_postconditions(Adaptor mesh, Matrix A, Vector Bu, Vector Bv) [protected, virtual]*

Check parameterize() postconditions: 3D -> 2D mapping is one-to-one.

bool *param.is_one_to_one_mapping(Adaptor mesh, Matrix A, Vector Bu, Vector Bv) [protected, virtual]*

Check if 3D -> 2D mapping is one-to-one. The default implementation checks each normal.

Border_param& *param.get_border_parameterizer()*

Get the object that maps the surface's border onto a 2D space.

Sparse_LA& *param.get_linear_algebra_traits()*

Get the sparse linear algebra (traits object to access the linear system).

See Also

CGAL::Parameterizer_traits_3<ParameterizationMesh_3> . . . . . . page 3393
CGAL::Barycentric_mapping_parameterizer_3<ParameterizationMesh_3, BorderParameterizer_3, SparseLinearAlgebraTraits_d> . . . . . . page 3350
CGAL::Discrete_authalic_parameterizer_3<ParameterizationMesh_3, BorderParameterizer_3, SparseLinearAlgebraTraits_d> . . . . . . page 3358
CGAL::Discrete_conformal_map_parameterizer_3<ParameterizationMesh_3, BorderParameterizer_3, SparseLinearAlgebraTraits_d> . . . . . . page 3360
CGAL::LSCM_parameterizer_3<ParameterizationMesh_3, BorderParameterizer_3, SparseLinearAlgebraTraits_d> . . . . . . page 3365
CGAL::Mean_value_coordinates_parameterizer_3<ParameterizationMesh_3, BorderParameterizer_3, SparseLinearAlgebraTraits_d> . . . . . . page 3369

CGAL::LSCM_parameterizer_3<ParameterizationMesh_3, BorderParameterizer_3, SparseLinearAlgebraTraits_d>

Class

Definition

The class *LSCM_parameterizer_3* implements the Least Squares Conformal Maps (LSCM) parameterization [LPRM02].

This is a conformal parameterization, i.e. it attempts to preserve angles.

This is a free border parameterization. No need to map the surface's border onto a convex polygon (only two pinned vertices are needed to ensure a unique solution), but one-to-one mapping is **not** guaranteed.

#include <CGAL/LSCM_parameterizer_3.h>

Inherits From

Parameterizer_traits_3<ParameterizationMesh_3>

Is Model for the Concepts

Model of the *ParameterizerTraits_3* concept.

Design Pattern

LSCM_parameterizer_3 class is a Strategy [GHJV95]: it implements a strategy of surface parameterization for models of *ParameterizationMesh_3*.

Parameters

The full template declaration is:

template<class *ParameterizationMesh_3*, class *BorderParameterizer_3 = Two_vertices_parameterizer_3< ParameterizationMesh_3>*, class *SparseLinearAlgebraTraits_d = OpenNL::SymmetricLinearSolverTraits< typename ParameterizationMesh_3::NT>>*
class *LSCM_parameterizer_3*;

Types

LSCM_parameterizer_3<ParameterizationMesh_3, BorderParameterizer_3, SparseLinearAlgebraTraits_d>:: Border_param

Export *BorderParameterizer_3* template parameter.

LSCM_parameterizer_3<ParameterizationMesh_3, BorderParameterizer_3, SparseLinearAlgebraTraits_d>:: Sparse_LA

Export *SparseLinearAlgebraTraits_d* template parameter.

Creation

LSCM_parameterizer_3<ParameterizationMesh_3, BorderParameterizer_3, SparseLinearAlgebraTraits_d> param(Border_param border_param = Border_param(), *Sparse_*

LA sparse_la = Sparse_LA())

Constructor.
Parameters:
border_param: Object that maps the surface's border to 2D space. *sparse_la*: Traits object to access a sparse linear system.

Operations

Error_code *param.parameterize(Adaptor& mesh)*

Compute a one-to-one mapping from a triangular 3D surface *mesh* to a piece of the 2D space. The mapping is linear by pieces (linear in each triangle). The result is the (u, v) pair image of each vertex of the 3D surface.
Preconditions:
mesh must be a surface with one connected component. *mesh* must be a triangular mesh.

See Also

CGAL::Parameterizer_traits_3<ParameterizationMesh_3> . . . . page 3393
CGAL::Fixed_border_parameterizer_3<ParameterizationMesh_3, BorderParameterizer_3, SparseLinearAlgebraTraits_d> . . . . page 3362
CGAL::Barycentric_mapping_parameterizer_3<ParameterizationMesh_3, BorderParameterizer_3, SparseLinearAlgebraTraits_d> . . . . page 3350
CGAL::Discrete_authalic_parameterizer_3<ParameterizationMesh_3, BorderParameterizer_3, SparseLinearAlgebraTraits_d> . . . . page 3358
CGAL::Discrete_conformal_map_parameterizer_3<ParameterizationMesh_3, BorderParameterizer_3, SparseLinearAlgebraTraits_d> . . . . page 3360
CGAL::Mean_value_coordinates_parameterizer_3<ParameterizationMesh_3, BorderParameterizer_3, SparseLinearAlgebraTraits_d> . . . . page 3369

SparseLinearAlgebraTraits_d::Matrix

Concept

Definition

SparseLinearAlgebraTraits_d::Matrix is a concept of a sparse matrix class.

Refines

LinearAlgebraTraits_d::Matrix

Types

Matrix:: NT

Creation

Matrix M(int dimension);	Create a square matrix initialized with zeros.
Matrix M(int rows, int columns);	Create a rectangular matrix initialized with zeros.

Operations

int *M.row_dimension() const*

Return the matrix number of rows.

int *M.column_dimension() const*

Return the matrix number of columns.

NT *M.get_coef(int row, int column) const*

Read access to a matrix coefficient.
Preconditions:
0 <= row < *row_dimension*(). 0 <= column < *column_dimension*().

void *M.add_coef(int row, int column, NT value)*

Write access to a matrix coefficient: *a_ij* <- *a_ij* + val.
Preconditions:
0 <= row < *row_dimension*(). 0 <= column < *column_dimension*().

void *M.set_coef(int row, int column, NT value, bool new_coef = false)*

Write access to a matrix coefficient: *a_ij* <- val.
Optimization: Caller can optimize this call by setting *new_coef* to true if the coefficient does not already exist in the matrix.
Preconditions:
0 <= i < *row_dimension*(). 0 <= j < *column_dimension*().

Has Models

Eigen_sparse_matrix<T>
Eigen_sparse_symmetric_matrix<T>
OpenNL::SparseMatrix<T> in *OpenNL* package

See Also

SparseLinearAlgebraTraits_d . . . . page 3400
SparseLinearAlgebraTraits_d::Vector . . . . page 3414

CGAL::Mean_value_coordinates_parameterizer_3< ParameterizationMesh_3, BorderParameterizer_3, SparseLinearAlgebraTraits_d>

Definition

The class *Mean_value_coordinates_parameterizer_3* implements Floater Mean Value Coordinates parameterization [Flo03a]. This method is sometimes called simply *Floater parameterization*.

This is a conformal parameterization, i.e. it attempts to preserve angles.

One-to-one mapping is guaranteed if the surface's border is mapped to a convex polygon.

This class is a Strategy [GHJV95] called by the main parameterization algorithm *Fixed_border_parameterizer_3::parameterize*(). It provides default *BorderParameterizer_3* and *SparseLinearAlgebraTraits_d* template parameters that make sense. It implements *compute_w_ij*() to compute *w_ij* = (i, j) coefficient of matrix A for j neighbor vertex of i based on Floater Mean Value Coordinates parameterization. It implements an optimized version of *is_one_to_one_mapping*().

#include <CGAL/Mean_value_coordinates_parameterizer_3.h>

Inherits From

Fixed_border_parameterizer_3<ParameterizationMesh_3, BorderParameterizer_3, SparseLinearAlgebraTraits_d>

Is Model for the Concepts

Model of the *ParameterizerTraits_3* concept.

Design Pattern

Mean_value_coordinates_parameterizer_3 class is a Strategy [GHJV95]: it implements a strategy of surface parameterization for models of *ParameterizationMesh_3*.

Parameters

The full template declaration is:

template<class *ParameterizationMesh_3*, class *BorderParameterizer_3* = *Circular_border_arc_length_parameterizer_3<ParameterizationMesh_3>*, class *SparseLinearAlgebraTraits_d* = *OpenNL::DefaultLinearSolverTraits<typename ParameterizationMesh_3::NT>*>
class *Mean_value_coordinates_parameterizer_3*;

Creation

Mean_value_coordinates_parameterizer_3<ParameterizationMesh_3, BorderParameterizer_3, SparseLinearAlgebraTraits_d> param(Border_param border_param = Border_param(),

LA sparse_la = Sparse_LA())

Constructor.
Parameters:
border_param: Object that maps the surface's border to 2D space. *sparse_la*: Traits object to access a sparse linear system.

Operations

virtual NT *param.compute_w_ij(Adaptor mesh, Vertex_const_handle main_vertex_v_i, Vertex_around_vertex_const_circulator neighbor_vertex_v_j)*
[protected, virtual]

Compute *w_ij* = (i, j) coefficient of matrix A for j neighbor vertex of i.

virtual bool *param.is_one_to_one_mapping(Adaptor, Matrix, Vector, Vector) [protected, virtual]*

Check if 3D -> 2D mapping is one-to-one.
Theorem: one-to-one mapping is guaranteed if all *w_ij* coefficients are > 0 (for j vertex neighbor of i) and if the surface border is mapped onto a 2D convex polygon. Floater formula above implies that *w_ij* > 0 (for j vertex neighbor of i), thus mapping is guaranteed if the surface border is mapped onto a 2D convex polygon.

See Also

CGAL::Parameterizer_traits_3<ParameterizationMesh_3> . . . . . . page 3393
CGAL::Fixed_border_parameterizer_3<ParameterizationMesh_3, BorderParameterizer_3, SparseLinearAlgebraTraits_d> . . . . . . page 3362
CGAL::Barycentric_mapping_parameterizer_3<ParameterizationMesh_3, BorderParameterizer_3, SparseLinearAlgebraTraits_d> . . . . . . page 3350
CGAL::Discrete_authalic_parameterizer_3<ParameterizationMesh_3, BorderParameterizer_3, SparseLinearAlgebraTraits_d> . . . . . . page 3358
CGAL::Discrete_conformal_map_parameterizer_3<ParameterizationMesh_3, BorderParameterizer_3, SparseLinearAlgebraTraits_d> . . . . . . page 3360
CGAL::LSCM_parameterizer_3<ParameterizationMesh_3, BorderParameterizer_3, SparseLinearAlgebraTraits_d> . . . . . . page 3365

Example

See *Simple_parameterization.cpp* example.

Concept

ParameterizationMesh_3

Definition

ParameterizationMesh_3 is a concept for a 3D surface mesh. Its main purpose is to allow the parameterization methods to access meshes in a uniform manner.

A *ParameterizationMesh_3* surface consists of vertices, facets and an incidence relation on them. No notion of edge is requested. Vertices represent points in 3d-space. Facets are planar polygons without holes defined by the circular sequence of vertices along their border. The surface itself can have holes. The vertices along the border of a hole are called *border vertices*. A surface is *closed* if it contains no border vertices.

The surface must be an oriented 2-manifold with border vertices, i.e. the neighborhood of each point on the surface is either homeomorphic to a disc or to a half disc, except for vertices where many holes and surfaces with border can join.

ParameterizationMesh_3 defines the types, data and methods that a mesh must implement to allow surface parameterization. Among other things, this concept defines accessors to fields specific to parameterizations methods: index, u, v, *is_parameterized*.

ParameterizationMesh_3 meshes can have any genus, arity or number of components. On the other hand, as parameterization methods deal only with topological disks, *ParameterizationMesh_3* defines an interface oriented towards topological disks.

Design Pattern

ParameterizationMesh_3 is an Adaptor [GHJV95]: it changes the interface of a 3D mesh to match the interface expected by the parameterization methods.

Types

The following mutable handles, iterators, and circulators have appropriate non-mutable counterparts, i.e. *const_handle*, *const_iterator*, and *const_circulator*. The mutable types are assignable to their non-mutable counterparts. Both circulators are assignable to the *Vertex_iterator*. The iterators are assignable to the respective handle types. Wherever the handles appear in function parameter lists, the corresponding iterators can be used as well.

ParameterizationMesh_3:: NT	Number type to represent coordinates.
ParameterizationMesh_3:: Point_2	2D point that represents (u, v) coordinates computed by parameterization methods. Must provide X() and Y() methods.
ParameterizationMesh_3:: Point_3	3D point that represents vertices coordinates. Must provide X() and Y() methods.
ParameterizationMesh_3:: Vector_2	2D vector. Must provide X() and Y() methods.
ParameterizationMesh_3:: Vector_3	3D vector. Must provide X() and Y() methods.
ParameterizationMesh_3:: Facet	Opaque type representing a facet of the 3D mesh. No methods are expected.
ParameterizationMesh_3:: Facet_handle	Handle to a facet. Model of the Handle concept.
ParameterizationMesh_3:: Facet_const_handle	
ParameterizationMesh_3:: Facet_iterator	Iterator over all mesh facets. Model of the *ForwardIterator* concept.

ParameterizationMesh_3:: Facet_const_iterator	
ParameterizationMesh_3:: Vertex	Opaque type representing a vertex of the 3D mesh. No methods are expected.
ParameterizationMesh_3:: Vertex_handle	Handle to a vertex. Model of the Handle concept.
ParameterizationMesh_3:: Vertex_const_handle	
ParameterizationMesh_3:: Vertex_iterator	Iterator over all vertices of a mesh. Model of the *ForwardIterator* concept.
ParameterizationMesh_3:: Vertex_const_iterator	
ParameterizationMesh_3:: Border_vertex_iterator	Iterator over vertices of the mesh *main border*. Model of the *ForwardIterator* concept.
ParameterizationMesh_3:: Border_vertex_const_iterator	
ParameterizationMesh_3:: Vertex_around_facet_circulator	Counter-clockwise circulator over a facet's vertices. Model of the *BidirectionalCirculator* concept.
ParameterizationMesh_3:: Vertex_around_facet_const_circulator	
ParameterizationMesh_3:: Vertex_around_vertex_circulator	Clockwise circulator over the vertices incident to a vertex. Model of the *BidirectionalCirculator* concept.
ParameterizationMesh_3:: Vertex_around_vertex_const_circulator	

Creation

Construction and destruction are undefined.

Operations

The following mutable methods returning a handle, iterator, or circulator have appropriate non-mutable counterpart methods, i.e. *const*, returning a *const_handle*, *const_iterator*, or *const_circulator*.

bool	*mesh.is_valid() const*	Indicate if the mesh matches the *ParameterizationMesh_3* concept.
Vertex_iterator	*mesh.mesh_vertices_begin()*	Get iterator over first vertex of mesh.
Vertex_const_iterator	*mesh.mesh_vertices_begin() const*	
Vertex_iterator	*mesh.mesh_vertices_end()*	Get iterator over past-the-end vertex of mesh.
Vertex_const_iterator	*mesh.mesh_vertices_end() const*	
int	*mesh.count_mesh_vertices() const*	Count the number of vertices of the mesh.

void *mesh.index_mesh_vertices()*

Index vertices of the mesh from 0 to *count_mesh_vertices*()-1.

Border_vertex_iterator

mesh.mesh_main_border_vertices_begin()

Get iterator over first vertex of mesh's *main border*.

Border_vertex_const_iterator

mesh.mesh_main_border_vertices_begin() const

Border_vertex_iterator

mesh.mesh_main_border_vertices_end()

Get iterator over past-the-end vertex of mesh's *main border*.

Border_vertex_const_iterator

mesh.mesh_main_border_vertices_end() const

std::list<Vertex_handle>

mesh.get_border(Vertex_handle seed_vertex)

Return the border containing *seed_vertex*. Return an empty list if not found.

Facet_iterator *mesh.mesh_facets_begin()*

Get iterator over first facet of mesh.

Facet_const_iterator *mesh.mesh_facets_begin() const*

Facet_iterator *mesh.mesh_facets_end()*

Get iterator over past-the-end facet of mesh.

Facet_const_iterator *mesh.mesh_facets_end() const*

int *mesh.count_mesh_facets() const*

Count the number of facets of the mesh.

bool *mesh.is_mesh_triangular() const*

Return true of all mesh's facets are triangles.

int *mesh.count_mesh_halfedges() const*

Count the number of halfedges of the mesh.

Vertex_around_facet_circulator

mesh.facet_vertices_begin(Facet_handle facet)

Get circulator over facet's vertices.

Vertex_around_facet_const_circulator

mesh.facet_vertices_begin(Facet_const_handle facet) const *int* *mesh.count_facet_vertices(Facet_const_handle facet) const*

Count the number of vertices of a facet.

Point_3 *mesh.get_vertex_position(Vertex_const_handle vertex) const*

Get the 3D position of a vertex.

Point_2 *mesh.get_vertex_uv(Vertex_const_handle vertex) const*

Get/set the 2D position (u/v pair) of a vertex. Default value is undefined.

void *mesh.set_vertex_uv(Vertex_handle vertex, Point_2 uv)*

bool *mesh.is_vertex_parameterized(Vertex_const_handle vertex) const*

Get/set *is parameterized* field of vertex. Default value is undefined.

void *mesh.set_vertex_parameterized(Vertex_handle vertex, bool parameterized)*

int *mesh.get_vertex_index(Vertex_const_handle vertex) const*

Get/set vertex index. Default value is undefined.

void *mesh.set_vertex_index(Vertex_handle vertex, int index)*

int *mesh.get_vertex_tag(Vertex_const_handle vertex) const*

Get/set vertex' all purpose tag. Default value is undefined.

void *mesh.set_vertex_tag(Vertex_handle vertex, int tag)*

bool *mesh.is_vertex_on_border(Vertex_const_handle vertex) const*

Return true if a vertex belongs to ANY mesh's border.

bool *mesh.is_vertex_on_main_border(Vertex_const_handle vertex) const*

Return true if a vertex belongs to the UNIQUE mesh's main border.

Vertex_around_vertex_circulator

mesh.vertices_around_vertex_begin(Vertex_handle vertex, Vertex_handle start_position = Vertex_handle())

Get circulator over the vertices incident to *vertex*. *start_position* defines the optional initial position of the circulator.

Vertex_around_vertex_const_circulator

mesh.vertices_around_vertex_begin(Vertex_const_handle vertex, Vertex_const_handle start_position = Vertex_const_handle())

Has Models

We provide 2 models of this concept:

Parameterization_polyhedron_adaptor_3<Polyhedron_3_>
Parameterization_mesh_patch_3<ParameterizationPatchableMesh_3>

See Also

ParameterizationPatchableMesh_3 . . . . page 3383

CGAL::Parameterization_mesh_feature_extractor< ParameterizationMesh_3>

Class

Definition

The class *Parameterization_mesh_feature_extractor* computes features (genus, borders, ...) of a 3D surface, model of the *ParameterizationMesh_3* concept.

#include <CGAL/Parameterization_mesh_feature_extractor.h>

Parameters

The full template declaration is:

template<class *ParameterizationMesh_3*>
class *Parameterization_mesh_feature_extractor*;

Types

Parameterization_mesh_feature_extractor<ParameterizationMesh_3>:: Adaptor

Export *ParameterizationMesh_3* template parameter.

Parameterization_mesh_feature_extractor<ParameterizationMesh_3>:: Border

Type representing a border = STL container of vertex handles.

Parameterization_mesh_feature_extractor<ParameterizationMesh_3>:: Skeleton

Type representing the list of all borders of the mesh = STL container of Border elements.

Creation

Parameterization_mesh_feature_extractor<ParameterizationMesh_3> extractor(Adaptor& mesh);

Constructor.
CAUTION: This class caches the result of feature extractions => The caller must **not** modify *mesh* during the *Parameterization_mesh_feature_extractor* life cycle.

Operations

int *extractor.get_nb_borders()*

Get number of borders.

Skeleton *extractor.get_borders()*

Get extracted borders. The longest border is the first one.

Border	*extractor.get_longest_border()*	
		Get longest border.
int	*extractor.get_nb_connex_components()*	
		Get # of connected components.
int	*extractor.get_genus()*	Get the genus.

See Also

ParameterizationMesh_3 . . . . page 3371

Example

See *Mesh_cutting_parameterization.cpp* example.

CGAL::Parameterization_mesh_patch_3<ParameterizationPatchableMesh_3>

Class

Definition

Parameterization_mesh_patch_3 is a Decorator class to *virtually* cut a patch in a *ParameterizationPatchableMesh_3* 3D surface. Only the patch is exported, making the 3D surface look like a topological disk.

The input mesh can be of any genus, but it has to come with a *seam* that describes the border of a topological disc. This border may be an actual border of the mesh or a virtual border.

#include <CGAL/Parameterization_mesh_patch_3.h>

Is Model for the Concepts

Model of *ParameterizationMesh_3* concept, whose purpose is to allow the *Surface_mesh_parameterization* package to access meshes in a uniform manner.

Design Pattern

Parameterization_mesh_patch_3 is a Decorator [GHJV95]: it changes the behavior of a *ParameterizationPatchableMesh_3* 3D surface while keeping its *ParameterizationMesh_3* interface.

Parameters

The full template declaration is:

template<class *ParameterizationPatchableMesh_3*>
class *Parameterization_mesh_patch_3*;

Parameters:
ParameterizationPatchableMesh_3: 3D surface mesh.

Types

The following mutable handles, iterators, and circulators have appropriate non-mutable counterparts, i.e. *const_handle*, *const_iterator*, and *const_circulator*. The mutable types are assignable to their non-mutable counterparts. Both circulators are assignable to the *Vertex_iterator*. The iterators are assignable to the respective handle types. Wherever the handles appear in function parameter lists, the corresponding iterators can be used as well.

Parameterization_mesh_patch_3<ParameterizationPatchableMesh_3>:: Adaptor

Export template parameter.

Types implementing the ParameterizationMesh_3 interface

Parameterization_mesh_patch_3<ParameterizationPatchableMesh_3>:: NT

Number type to represent coordinates.

Parameterization_mesh_patch_3<ParameterizationPatchableMesh_3>:: Point_2

2D point that represents (u, v) coordinates computed by parameterization methods. Must provide X() and Y() methods.

Parameterization_mesh_patch_3<ParameterizationPatchableMesh_3>:: Point_3

3D point that represents vertices coordinates. Must provide X() and Y() methods.

Parameterization_mesh_patch_3<ParameterizationPatchableMesh_3>:: Vector_2

2D vector. Must provide X() and Y() methods.

Parameterization_mesh_patch_3<ParameterizationPatchableMesh_3>:: Vector_3

3D vector. Must provide X() and Y() methods.

Parameterization_mesh_patch_3<ParameterizationPatchableMesh_3>:: Facet

Opaque type representing a facet of the 3D mesh. No methods are expected.

Parameterization_mesh_patch_3<ParameterizationPatchableMesh_3>:: Facet_handle

Handle to a facet. Model of the Handle concept.

Parameterization_mesh_patch_3<ParameterizationPatchableMesh_3>:: Facet_const_handle
Parameterization_mesh_patch_3<ParameterizationPatchableMesh_3>:: Facet_iterator

Iterator over all mesh facets. Model of the *ForwardIterator* concept.

Parameterization_mesh_patch_3<ParameterizationPatchableMesh_3>:: Facet_const_iterator
Parameterization_mesh_patch_3<ParameterizationPatchableMesh_3>:: Vertex

Opaque type representing a vertex of the 3D mesh. No methods are expected.

Parameterization_mesh_patch_3<ParameterizationPatchableMesh_3>:: Vertex_handle

Handle to a vertex. Model of the Handle concept.

Parameterization_mesh_patch_3<ParameterizationPatchableMesh_3>:: Vertex_const_handle
Parameterization_mesh_patch_3<ParameterizationPatchableMesh_3>:: Vertex_iterator

Iterator over all vertices of a mesh. Model of the *ForwardIterator* concept.

Parameterization_mesh_patch_3<ParameterizationPatchableMesh_3>:: Vertex_const_iterator
Parameterization_mesh_patch_3<ParameterizationPatchableMesh_3>:: Border_vertex_iterator

Iterator over vertices of the mesh *main border*. Model of the *ForwardIterator* concept.

Parameterization_mesh_patch_3<ParameterizationPatchableMesh_3>:: Border_vertex_const_iterator

Parameterization_mesh_patch_3<ParameterizationPatchableMesh_3>:: Vertex_around_facet_circulator

Counter-clockwise circulator over a facet's vertices. Model of the *BidirectionalCirculator* concept.

Parameterization_mesh_patch_3<ParameterizationPatchableMesh_3>:: Vertex_around_facet_const_circulator
Parameterization_mesh_patch_3<ParameterizationPatchableMesh_3>:: Vertex_around_vertex_circulator

Clockwise circulator over the vertices incident to a vertex. Model of the *BidirectionalCirculator* concept.

Parameterization_mesh_patch_3<ParameterizationPatchableMesh_3>:: Vertex_around_vertex_const_circulator

Variables

Adaptor&	*m_mesh_adaptor;*	The decorated mesh.

Creation

template<class InputIterator>
Parameterization_mesh_patch_3<ParameterizationPatchableMesh_3> mesh(Adaptor& mesh, InputIterator first_seam_vertex, InputIterator end_seam_vertex)

Create a Decorator for an existing *ParameterizationPatchableMesh_3* mesh. The input mesh can be of any genus, but it has to come with a *seam* that describes the border of a topological disc. This border may be an actual border of the mesh or a virtual border.
Preconditions:
first_seam_vertex -> *end_seam_vertex* defines the outer seam, i.e. *Parameterization_mesh_patch_3* will export the *right* of the seam. The *seam* is given as a container of *Adaptor::Vertex_handle* elements. The *seam* is implicitely a loop. The first vertex should **not** be duplicated at the end.

Operations

The following methods returning a mutable handle, iterator, or circulator have appropriate non-mutable counterpart methods, i.e. *const*, returning a *const_handle*, *const_iterator*, or *const_circulator*.

Adaptor& *mesh.get_decorated_mesh()*

Returns: the decorated mesh.

Adaptor *mesh.get_decorated_mesh() const*

Methods implementing the ParameterizationMesh_3 interface

bool *mesh.is_valid() const* Indicate if the mesh matches the *ParameterizationMesh_3* concept.

Vertex_iterator *mesh.mesh_vertices_begin()*

Get iterator over first vertex of mesh.

Vertex_const_iterator

mesh.mesh_vertices_begin() const

Vertex_iterator *mesh.mesh_vertices_end()*

Get iterator over past-the-end vertex of mesh.

Vertex_const_iterator

mesh.mesh_vertices_end() const

int *mesh.count_mesh_vertices() const*

Count the number of vertices of the mesh.

void *mesh.index_mesh_vertices()*

Index vertices of the mesh from 0 to *count_mesh_vertices*()-1.

Border_vertex_iterator

mesh.mesh_main_border_vertices_begin()

Get iterator over first vertex of mesh's main border (aka *seam*).

Border_vertex_const_iterator

mesh.mesh_main_border_vertices_begin() const

Border_vertex_iterator

mesh.mesh_main_border_vertices_end()

Get iterator over past-the-end vertex of mesh's main border (aka *seam*).

Border_vertex_const_iterator

mesh.mesh_main_border_vertices_end() const

std::list<Vertex_handle>

mesh.get_border(Vertex_handle seed_vertex)

Returns: the border containing *seed_vertex* (or an empty list if not found).
Parameters:
seed_vertex: a border vertex.

Facet_iterator *mesh.mesh_facets_begin()*

Get iterator over first facet of mesh.

Facet_const_iterator *mesh.mesh_facets_begin() const*

Facet_iterator *mesh.mesh_facets_end()*

Get iterator over past-the-end facet of mesh.

Facet_const_iterator *mesh.mesh_facets_end() const*

int *mesh.count_mesh_facets() const*

Count the number of facets of the mesh.

bool *mesh.is_mesh_triangular() const*

Return true of all mesh's facets are triangles.

int *mesh.count_mesh_halfedges() const*

Count the number of halfedges of the mesh.

Vertex_around_facet_circulator

mesh.facet_vertices_begin(Facet_handle facet)

Get circulator over facet's vertices.

Vertex_around_facet_const_circulator

mesh.facet_vertices_begin(Facet_const_handle facet) const

int *mesh.count_facet_vertices(Facet_const_handle facet) const*

Count the number of vertices of a facet.

Point_3 *mesh.get_vertex_position(Vertex_const_handle vertex) const*

Get the 3D position of a vertex.

Point_2 *mesh.get_vertex_uv(Vertex_const_handle vertex) const*

Get/set the 2D position (u/v pair) of a vertex. Default value is undefined.

void *mesh.set_vertex_uv(Vertex_handle vertex, Point_2 uv)*

bool *mesh.is_vertex_parameterized(Vertex_const_handle vertex) const*

Get/set *is parameterized* field of vertex. Default value is undefined.

void *mesh.set_vertex_parameterized(Vertex_handle vertex, bool parameterized)*

int *mesh.get_vertex_index(Vertex_const_handle vertex) const*

Get/set vertex index. Default value is undefined.

void *mesh.set_vertex_index(Vertex_handle vertex, int index)*

int *mesh.get_vertex_tag(Vertex_const_handle vertex) const*

Get/set vertex' all purpose tag. Default value is undefined.

void *mesh.set_vertex_tag(Vertex_handle vertex, int tag)*

bool *mesh.is_vertex_on_border(Vertex_const_handle vertex) const*

Return true if a vertex belongs to ANY mesh's border.

bool *mesh.is_vertex_on_main_border(Vertex_const_handle vertex) const*

Return true if a vertex belongs to the UNIQUE mesh's main border set by the constructor.

Vertex_around_vertex_circulator

mesh.vertices_around_vertex_begin(Vertex_handle vertex, Vertex_handle start_position = Vertex_handle())

Get circulator over the vertices incident to *vertex*. *start_position* defines the optional initial position of the circulator.

Vertex_around_vertex_const_circulator

mesh.vertices_around_vertex_begin(Vertex_const_handle vertex, Vertex_const_handle start_position = Vertex_const_handle())

See Also

CGAL::Parameterization_polyhedron_adaptor_3<Polyhedron_3_> . . . . page 3386

Example

See *Mesh_cutting_parameterization.cpp* example.

Concept

ParameterizationPatchableMesh_3

Definition

ParameterizationPatchableMesh_3 inherits from concept *ParameterizationMesh_3*, thus is a concept of a 3D surface mesh.

ParameterizationPatchableMesh_3 adds the ability to support patches and virtual seams. *Patches* are a subset of a 3D mesh. *Virtual seams* are the ability to behave exactly as if the surface was cut following a certain path.

This mainly means that: vertices can be tagged as inside or outside the patch to parameterize. the fields specific to parameterizations (index, u, v, *is_parameterized*) can be set per *corner* (aka half-edge).

The main purpose of this feature is to allow the *Surface_mesh_parameterization* package to parameterize any 3D surface by decomposing it as a list of topological disks.

Design Pattern

ParameterizationPatchableMesh_3 is an Adaptor [GHJV95]: it changes the interface of a 3D mesh to match the interface expected by class *Parameterization_mesh_patch_3*.

Refines

ParameterizationMesh_3

In addition to the requirements described in the concept *ParameterizationMesh_3*, *ParameterizationPatchableMesh_3* provides the following:

Creation

Construction and destruction are undefined.

Operations

int	*mesh.get_vertex_seaming(Vertex_const_handle vertex) const*	
		Get/set vertex seaming flag. Default value is undefined.
void	*mesh.set_vertex_seaming(Vertex_handle vertex, int seaming)*	
int	*mesh.get_halfedge_seaming(Vertex_const_handle source, Vertex_const_handle target) const*	
		Get/set oriented edge's seaming flag, i.e. position of the oriented edge w.r.t. to the UNIQUE main border.
void	*mesh.set_halfedge_seaming(Vertex_handle source, Vertex_handle target, int seaming)*	

Point_2	*mesh.get_corners_uv(Vertex_const_handle vertex, Vertex_const_handle prev_vertex, Vertex_const_handle next_vertex) const*	
		Get/set the 2D position (= (u, v) pair) of corners at the *right* of the *prev_vertex* -> vertex -> *next_vertex* line. Default value is undefined.
void	*mesh.set_corners_uv(Vertex_handle vertex, Vertex_const_handle prev_vertex, Vertex_const_handle next_vertex, Point_2 uv)*	
bool	*mesh.are_corners_parameterized(Vertex_const_handle vertex, Vertex_const_handle prev_vertex, Vertex_const_handle next_vertex) const*	
		Get/set *is parameterized* field of corners at the *right* of the *prev_vertex* -> vertex -> *next_vertex* line. Default value is undefined.
void	*mesh.set_corners_parameterized(Vertex_handle vertex, Vertex_const_handle prev_vertex, Vertex_const_handle next_vertex, bool parameterized)*	
int	*mesh.get_corners_index(Vertex_const_handle vertex, Vertex_const_handle prev_vertex, Vertex_const_handle next_vertex) const*	
		Get/set index of corners at the *right* of the *prev_vertex* -> vertex -> *next_vertex* line. Default value is undefined.
void	*mesh.set_corners_index(Vertex_handle vertex, Vertex_const_handle prev_vertex, Vertex_const_handle next_vertex, int index)*	
int	*mesh.get_corners_tag(Vertex_const_handle vertex, Vertex_const_handle prev_vertex, Vertex_const_handle next_vertex) const*	
		Get/set all purpose tag of corners at the *right* of the *prev_vertex* -> vertex -> *next_vertex* line. Default value is undefined.
void	*mesh.set_corners_tag(Vertex_handle vertex, Vertex_const_handle prev_vertex, Vertex_const_handle next_vertex, int tag)*	

Has Models

Adaptator for *Polyhedron_3* is provided.

CGAL::Parameterization_polyhedron_adaptor_3<Polyhedron_3_> . . . . page 3386

See Also

ParameterizationMesh_3 . . . . page 3371

Concept

PolyhedronTraitsWithKernel_3

Required types for the PolyhedronTraitsWithKernel_3 concept. This geometric traits concept is used in the polyhedral surface data structure *CGAL::Polyhedron_3<Traits>*. This concept provides additional requirements to the *PolyhedronTraits_3* concept required by *CGAL::Polyhedron_3<Traits>* used within the class *Parameterization_polyhedron_adaptor_3<Polyhedron_3_>*.

Refines

PolyhedronTraits_3.

Types

PolyhedronTraitsWithKernel_3:: Kernel	a kernel type providing: a field type *FT*; 2D and 3D point types *Point_2* and *Point_3*; 2D and 3D vector types *Vector_2* and *Vector_3*.

Has Models

CGAL::Polyhedron_traits_3<Kernel> . . . . page 1824
CGAL::Polyhedron_traits_with_normals_3<Kernel> . . . . page 1826
All models of the *CGAL::Kernel* concept, e.g., *Simple_cartesian<FieldNumberType>*.

See Also

CGAL::Parameterization_polyhedron_adaptor_3<Polyhedron_3_> . . . . page 3386
CGAL::Polyhedron_3<Traits> . . . . page 1795

Class

CGAL::Parameterization_polyhedron_adaptor_3<Polyhedron_3_>

Definition

Parameterization_polyhedron_adaptor_3 is an adaptor class to access to a Polyhedron 3D mesh using the *ParameterizationPatchableMesh_3* interface. Among other things, this concept defines the accessor to the (u, v) values computed by parameterizations methods.

Note that these interfaces are decorators that add *on the fly* the necessary fields to unmodified CGAL data structures (using STL maps). For performance reasons, it is recommended to use CGAL data structures enriched with the proper fields.

A *ParameterizationMesh_3* surface consists of vertices, facets and an incidence relation on them. No notion of edge is requested.

ParameterizationMesh_3 meshes can have any genus, arity or number of components.

It can have have any number of borders. Its *main border* will be the mesh's longest border (if there is at least one border).

It has also the ability to support patches and virtual seams. *Patches* are a subset of a 3D mesh. *Virtual seams* are the ability to behave exactly as if the surface was cut following a certain path.

#include <CGAL/Parameterization_polyhedron_adaptor_3.h>

Is Model for the Concepts

Model of *ParameterizationPatchableMesh_3* concept, whose purpose is to allow the *Surface_mesh_parameterization* package to access meshes in a uniform manner.

Design Pattern

Parameterization_polyhedron_adaptor_3 is an Adaptor [GHJV95]: it changes the Polyhedron interface to match the *ParameterizationPatchableMesh_3* concept.

Parameters

The full template declaration is:

template<class *Polyhedron_3_*>
class *Parameterization_polyhedron_adaptor_3*;

*Polyhedron_3_* must be of type *CGAL::Polyhedron<Traits>* and *Traits* must be a model of the concept *PolyhedronTraitsWithKernel_3*.

Types

The following mutable handles, iterators, and circulators have appropriate non-mutable counterparts, i.e. *const_handle*, *const_iterator*, and *const_circulator*. The mutable types are assignable to their non-mutable counterparts. Both circulators are assignable to the *Vertex_iterator*. The iterators are assignable to the respective handle types. Wherever the handles appear in function parameter lists, the corresponding iterators can be used as well.

Parameterization_polyhedron_adaptor_3<Polyhedron_3_>:: Polyhedron

Export template parameter.

Parameterization_polyhedron_adaptor_3<Polyhedron_3_>:: NT

Number type to represent coordinates.

Parameterization_polyhedron_adaptor_3<Polyhedron_3_>:: Point_2

2D point that represents (u, v) coordinates computed by parameterization methods. Must provide X() and Y() methods.

Parameterization_polyhedron_adaptor_3<Polyhedron_3_>:: Point_3

3D point that represents vertices coordinates. Must provide X() and Y() methods.

Parameterization_polyhedron_adaptor_3<Polyhedron_3_>:: Vector_2

2D vector. Must provide X() and Y() methods.

Parameterization_polyhedron_adaptor_3<Polyhedron_3_>:: Vector_3

3D vector. Must provide X() and Y() methods.

Parameterization_polyhedron_adaptor_3<Polyhedron_3_>:: Facet

Opaque type representing a facet of the 3D mesh. No methods are expected.

Parameterization_polyhedron_adaptor_3<Polyhedron_3_>:: Facet_handle

Handle to a facet. Model of the Handle concept.

Parameterization_polyhedron_adaptor_3<Polyhedron_3_>:: Facet_const_handle
Parameterization_polyhedron_adaptor_3<Polyhedron_3_>:: Facet_iterator

Iterator over all mesh facets. Model of the *ForwardIterator* concept.

Parameterization_polyhedron_adaptor_3<Polyhedron_3_>:: Facet_const_iterator
Parameterization_polyhedron_adaptor_3<Polyhedron_3_>:: Vertex

Opaque type representing a vertex of the 3D mesh. No methods are expected.

Parameterization_polyhedron_adaptor_3<Polyhedron_3_>:: Vertex_handle

Handle to a vertex. Model of the Handle concept.

Parameterization_polyhedron_adaptor_3<Polyhedron_3_>:: Vertex_const_handle
Parameterization_polyhedron_adaptor_3<Polyhedron_3_>:: Vertex_iterator

Iterator over all vertices of a mesh. Model of the *ForwardIterator* concept.

Parameterization_polyhedron_adaptor_3<Polyhedron_3_>:: Vertex_const_iterator
Parameterization_polyhedron_adaptor_3<Polyhedron_3_>:: Border_vertex_iterator

Iterator over vertices of the mesh *main border*. Model of the *ForwardIterator* concept.

Parameterization_polyhedron_adaptor_3<Polyhedron_3_>:: Border_vertex_const_iterator

Parameterization_polyhedron_adaptor_3<Polyhedron_3_>:: Vertex_around_facet_circulator

Counter-clockwise circulator over a facet's vertices. Model of the *BidirectionalCirculator* concept.

Parameterization_polyhedron_adaptor_3<Polyhedron_3_>:: Vertex_around_facet_const_circulator
Parameterization_polyhedron_adaptor_3<Polyhedron_3_>:: Vertex_around_vertex_circulator

Clockwise circulator over the vertices incident to a vertex. Model of the *BidirectionalCirculator* concept.

Parameterization_polyhedron_adaptor_3<Polyhedron_3_>:: Vertex_around_vertex_const_circulator

Creation

Parameterization_polyhedron_adaptor_3<Polyhedron_3_> mesh(Polyhedron& mesh);

Create an adaptator for an existing *Polyhedron_3* mesh. The input mesh can be of any genus. It can have have any number of borders. Its *main border* will be the mesh's longest border (if there is at least one border).

Operations

The following methods returning a mutable handle, iterator, or circulator have appropriate non-mutable counterpart methods, i.e. *const*, returning a *const_handle*, *const_iterator*, or *const_circulator*.

Polyhedron& *mesh.get_adapted_mesh()*

Get the adapted mesh.

Polyhedron *mesh.get_adapted_mesh() const*
Polyhedron::Halfedge_const_handle

mesh.get_halfedge(Vertex_const_handle source, Vertex_const_handle target) const

Get halfedge from source and target vertices. Will assert if such a halfedge doesn't exist.

Polyhedron::Halfedge_handle

mesh.get_halfedge(Vertex_handle source, Vertex_handle target)
*const Halfedge_info** *mesh.info(Halfedge_const_handle halfedge) const*

Access to additional info attached to halfedges.

*Halfedge_info** *mesh.info(Halfedge_const_handle halfedge)*
*const Vertex_info** *mesh.info(Vertex_const_handle vertex) const*

Access to additional info attached to vertices.

*Vertex_info** *mesh.info(Vertex_const_handle vertex)*
bool *mesh.is_valid() const* Indicate if the mesh matches the *ParameterizationMesh_3* concept.

Vertex_iterator *mesh.mesh_vertices_begin()*

Get iterator over first vertex of mesh.

Vertex_const_iterator

mesh.mesh_vertices_begin() const

Vertex_iterator *mesh.mesh_vertices_end()*

Get iterator over past-the-end vertex of mesh.

Vertex_const_iterator

mesh.mesh_vertices_end() const

int *mesh.count_mesh_vertices() const*

Count the number of vertices of the mesh.

void *mesh.index_mesh_vertices()*

Border_vertex_iterator

mesh.mesh_main_border_vertices_begin()

Get iterator over first vertex of mesh's *main border*.

Border_vertex_const_iterator

mesh.mesh_main_border_vertices_begin() const

Border_vertex_iterator

mesh.mesh_main_border_vertices_end()

Get iterator over past-the-end vertex of mesh's *main border*.

Border_vertex_const_iterator

mesh.mesh_main_border_vertices_end() const

std::list<Vertex_handle>

mesh.get_border(Vertex_handle seed_vertex)

Return the border containing *seed_vertex*. Return an empty list if not found.

Facet_iterator *mesh.mesh_facets_begin()*

Get iterator over first facet of mesh.

Facet_const_iterator *mesh.mesh_facets_begin() const*

Facet_iterator *mesh.mesh_facets_end()*

Get iterator over past-the-end facet of mesh.

Facet_const_iterator *mesh.mesh_facets_end() const*

int *mesh.count_mesh_facets() const*

Count the number of facets of the mesh.

bool *mesh.is_mesh_triangular() const*

Return true of all mesh's facets are triangles.

int *mesh.count_mesh_halfedges() const*

Count the number of halfedges of the mesh.

Vertex_around_facet_circulator

mesh.facet_vertices_begin(Facet_handle facet)

Get circulator over facet's vertices.

Vertex_around_facet_const_circulator

	mesh.facet_vertices_begin(Facet_const_handle facet) const	
int	*mesh.count_facet_vertices(Facet_const_handle facet) const*	
		Count the number of vertices of a facet.
Point_3	*mesh.get_vertex_position(Vertex_const_handle vertex) const*	
		Get the 3D position of a vertex.
Point_2	*mesh.get_vertex_uv(Vertex_const_handle vertex) const*	
		Get/set the 2D position (u/v pair) of a vertex. Default value is undefined. (stored in halfedges sharing the same vertex).
void	*mesh.set_vertex_uv(Vertex_handle vertex, Point_2 uv)*	
bool	*mesh.is_vertex_parameterized(Vertex_const_handle vertex) const*	
		Get/set *is parameterized* field of vertex. Default value is undefined. (stored in halfedges sharing the same vertex).
void	*mesh.set_vertex_parameterized(Vertex_handle vertex, bool parameterized)*	
int	*mesh.get_vertex_index(Vertex_const_handle vertex) const*	
		Get/set vertex index. Default value is undefined. (stored in Polyhedron vertex for debugging purpose).
void	*mesh.set_vertex_index(Vertex_handle vertex, int index)*	
int	*mesh.get_vertex_tag(Vertex_const_handle vertex) const*	
		Get/set vertex' all purpose tag. Default value is undefined. (stored in halfedges sharing the same vertex).
void	*mesh.set_vertex_tag(Vertex_handle vertex, int tag)*	
bool	*mesh.is_vertex_on_border(Vertex_const_handle vertex) const*	
		Return true if a vertex belongs to ANY mesh's border.
bool	*mesh.is_vertex_on_main_border(Vertex_const_handle vertex) const*	
		Return true if a vertex belongs to the UNIQUE mesh's main border, i.e. the mesh's LONGEST border.

Vertex_around_vertex_circulator

	mesh.vertices_around_vertex_begin(Vertex_handle vertex, Vertex_handle start_position = Vertex_handle())	
		Get circulator over the vertices incident to *vertex*. *start_position* defines the optional initial position of the circulator.

Vertex_around_vertex_const_circulator

	mesh.vertices_around_vertex_begin(Vertex_const_handle vertex, Vertex_const_handle start_position = Vertex_const_handle())	
int	*mesh.get_vertex_seaming(Vertex_const_handle vertex) const*	
		Get/set vertex seaming flag. Default value is undefined.
void	*mesh.set_vertex_seaming(Vertex_handle vertex, int seaming)*	

int	*mesh.get_halfedge_seaming(Vertex_const_handle source, Vertex_const_handle target) const*	
		Get/set oriented edge's seaming flag, i.e. position of the oriented edge w.r.t. to the UNIQUE main border.
void	*mesh.set_halfedge_seaming(Vertex_handle source, Vertex_handle target, int seaming)*	
Point_2	*mesh.get_corners_uv(Vertex_const_handle vertex, Vertex_const_handle prev_vertex, Vertex_const_handle next_vertex) const*	
		Get/set the 2D position (= (u, v) pair) of corners at the *right* of the *prev_vertex* -> vertex -> *next_vertex* line. Default value is undefined. (stored in incident halfedges).
void	*mesh.set_corners_uv(Vertex_handle vertex, Vertex_const_handle prev_vertex, Vertex_const_handle next_vertex, Point_2 uv)*	
bool	*mesh.are_corners_parameterized(Vertex_const_handle vertex, Vertex_const_handle prev_vertex, Vertex_const_handle next_vertex) const*	
		Get/set *is parameterized* field of corners at the *right* of the *prev_vertex* -> vertex -> *next_vertex* line. Default value is undefined. (stored in incident halfedges).
void	*mesh.set_corners_parameterized(Vertex_handle vertex, Vertex_const_handle prev_vertex, Vertex_const_handle next_vertex, bool parameterized)*	
int	*mesh.get_corners_index(Vertex_const_handle vertex, Vertex_const_handle prev_vertex, Vertex_const_handle next_vertex) const*	
		Get/set index of corners at the *right* of the *prev_vertex* -> vertex -> *next_vertex* line. Default value is undefined. (stored in incident halfedges).
void	*mesh.set_corners_index(Vertex_handle vertex, Vertex_const_handle prev_vertex, Vertex_const_handle next_vertex, int index)*	
int	*mesh.get_corners_tag(Vertex_const_handle vertex, Vertex_const_handle prev_vertex, Vertex_const_handle next_vertex) const*	
		Get/set all purpose tag of corners at the *right* of the *prev_vertex* -> vertex -> *next_vertex* line. Default value is undefined. (stored in incident halfedges).
void	*mesh.set_corners_tag(Vertex_handle vertex, Vertex_const_handle prev_vertex, Vertex_const_handle next_vertex, int tag)*	

See Also

CGAL::Parameterization_mesh_patch_3<ParameterizationPatchableMesh_3> . . . . page 3377

Example

See *Simple_parameterization.cpp* example.

CGAL::Parameterizer_traits_3<ParameterizationMesh_3>

Class

Definition

The class *Parameterizer_traits_3* is the base class of all parameterization methods. This class is a pure virtual class, thus cannot be instantiated.

This class doesn't do much. Its main goal is to ensure that subclasses will be proper models of the *ParameterizerTraits_3* concept: *Parameterizer_traits_3* defines the *Error_code* list of errors detected by this package. *Parameterizer_traits_3* declares a pure virtual method parameterize().

#include <CGAL/Parameterizer_traits_3.h>

Is Model for the Concepts

Model of the *ParameterizerTraits_3* concept (although you cannot instantiate this class).

Design Pattern

ParameterizerTraits_3 models are Strategies [GHJV95]: they implement a strategy of surface parameterization for models of *ParameterizationMesh_3*.

Parameters

The full template declaration is:

template<class *ParameterizationMesh_3*>
class *Parameterizer_traits_3*;

Types

Parameterizer_traits_3<ParameterizationMesh_3>:: Adaptor

Export *ParameterizationMesh_3* template parameter.

Parameterizer_traits_3<ParameterizationMesh_3>:: NT
Parameterizer_traits_3<ParameterizationMesh_3>:: Point_2
Parameterizer_traits_3<ParameterizationMesh_3>:: Point_3
Parameterizer_traits_3<ParameterizationMesh_3>:: Vector_2
Parameterizer_traits_3<ParameterizationMesh_3>:: Vector_3
Parameterizer_traits_3<ParameterizationMesh_3>:: Facet
Parameterizer_traits_3<ParameterizationMesh_3>:: Facet_handle
Parameterizer_traits_3<ParameterizationMesh_3>:: Facet_const_handle
Parameterizer_traits_3<ParameterizationMesh_3>:: Facet_iterator
Parameterizer_traits_3<ParameterizationMesh_3>:: Facet_const_iterator
Parameterizer_traits_3<ParameterizationMesh_3>:: Vertex
Parameterizer_traits_3<ParameterizationMesh_3>:: Vertex_handle
Parameterizer_traits_3<ParameterizationMesh_3>:: Vertex_const_handle
Parameterizer_traits_3<ParameterizationMesh_3>:: Vertex_iterator

Parameterizer_traits_3<ParameterizationMesh_3>:: Vertex_const_iterator
Parameterizer_traits_3<ParameterizationMesh_3>:: Border_vertex_iterator
Parameterizer_traits_3<ParameterizationMesh_3>:: Border_vertex_const_iterator
Parameterizer_traits_3<ParameterizationMesh_3>:: Vertex_around_facet_circulator
Parameterizer_traits_3<ParameterizationMesh_3>:: Vertex_around_facet_const_circulator
Parameterizer_traits_3<ParameterizationMesh_3>:: Vertex_around_vertex_circulator
Parameterizer_traits_3<ParameterizationMesh_3>:: Vertex_around_vertex_const_circulator

Constants

enum Error_code { *OK,*
ERROR_EMPTY_MESH,
ERROR_NON_TRIANGULAR_MESH,
ERROR_NO_TOPOLOGICAL_DISC,
ERROR_BORDER_TOO_SHORT,
ERROR_NON_CONVEX_BORDER,
ERROR_CANNOT_SOLVE_LINEAR_SYSTEM,
ERROR_NO_1_TO_1_MAPPING,
ERROR_OUT_OF_MEMORY,
ERROR_WRONG_PARAMETER}

List of errors detected by this package.
Values:
OK: Success. *ERROR_EMPTY_MESH*: Input mesh is empty. *ERROR_NON_TRIANGULAR_MESH*: Input mesh is not triangular. *ERROR_NO_TOPOLOGICAL_DISC*: Input mesh is not a topological disc. *ERROR_BORDER_TOO_SHORT*: This border parameterization requires a longer border. *ERROR_NON_CONVEX_BORDER*: This parameterization method requires a convex border. *ERROR_CANNOT_SOLVE_LINEAR_SYSTEM*: Cannot solve linear system. *ERROR_NO_1_TO_1_MAPPING*: Parameterization failed: no one-to-one mapping. *ERROR_OUT_OF_MEMORY*: Not enough memory. *ERROR_WRONG_PARAMETER*: A method received an unexpected parameter.

Creation

Operations

virtual Error_code *param.parameterize(Adaptor& mesh)*

Compute a one-to-one mapping from a 3D surface *mesh* to a piece of the 2D space. The mapping is linear by pieces (linear in each triangle). The result is the (u, v) pair image of each vertex of the 3D surface.
Preconditions:
mesh must be a surface with one connected component. *mesh* must be a triangular mesh.

*static const char** *param.get_error_message(int error_code)*

Get message (in English) corresponding to an error code
Parameters:
error_code: The code returned by parameterize().
Returns: The string describing the error code

See Also

CGAL::Fixed_border_parameterizer_3<ParameterizationMesh_3, BorderParameterizer_3, SparseLinearAlgebraTraits_d> . . . . page 3362
CGAL::Barycentric_mapping_parameterizer_3<ParameterizationMesh_3, BorderParameterizer_3, SparseLinearAlgebraTraits_d> . . . . page 3350
CGAL::Discrete_authalic_parameterizer_3<ParameterizationMesh_3, BorderParameterizer_3, SparseLinearAlgebraTraits_d> . . . . page 3358
CGAL::Discrete_conformal_map_parameterizer_3<ParameterizationMesh_3, BorderParameterizer_3, SparseLinearAlgebraTraits_d> . . . . page 3360
CGAL::LSCM_parameterizer_3<ParameterizationMesh_3, BorderParameterizer_3, SparseLinearAlgebraTraits_d> . . . . page 3365
CGAL::Mean_value_coordinates_parameterizer_3<ParameterizationMesh_3, BorderParameterizer_3, SparseLinearAlgebraTraits_d> . . . . page 3369

Concept

ParameterizerTraits_3

Definition

ParameterizerTraits_3 is a concept of parameterization object for a given type of mesh, *Adaptor*, which is a model of the *ParameterizationMesh_3* concept.

Design Pattern

ParameterizerTraits_3 models are Strategies [GHJV95]: they implement a strategy of surface parameterization for models of *ParameterizationMesh_3*.

Types

ParameterizerTraits_3:: Adaptor	Export the type of mesh to parameterize.

Constants

enum Error_code { *OK,*
ERROR_EMPTY_MESH,
ERROR_NON_TRIANGULAR_MESH,
ERROR_NO_TOPOLOGICAL_DISC,
ERROR_BORDER_TOO_SHORT,
ERROR_NON_CONVEX_BORDER,
ERROR_CANNOT_SOLVE_LINEAR_SYSTEM,
ERROR_NO_1_TO_1_MAPPING,
ERROR_OUT_OF_MEMORY,
ERROR_WRONG_PARAMETER}

List of errors detected by this package.
Values:
OK: Success. *ERROR_EMPTY_MESH*: Input mesh is empty. *ERROR_NON_TRIANGULAR_MESH*: Input mesh is not triangular. *ERROR_NO_TOPOLOGICAL_DISC*: Input mesh is not a topological disc. *ERROR_BORDER_TOO_SHORT*: This border parameterization requires a longer border. *ERROR_NON_CONVEX_BORDER*: This parameterization method requires a convex border. *ERROR_CANNOT_SOLVE_LINEAR_SYSTEM*: Cannot solve linear system. *ERROR_NO_1_TO_1_MAPPING*: Parameterization failed: no one-to-one mapping. *ERROR_OUT_OF_MEMORY*: Not enough memory. *ERROR_WRONG_PARAMETER*: A method received an unexpected parameter.

Creation

Construction and destruction are undefined.

Operations

Error_code *param.parameterize(Adaptor& mesh)*

Compute a one-to-one mapping from a triangular 3D surface *mesh* to a piece of the 2D space. The mapping is linear by pieces (linear in each triangle). The result is the (u, v) pair image of each vertex of the 3D surface.
Preconditions:
mesh must be a surface with one connected component and no hole. *mesh* must be a triangular mesh.

Has Models

CGAL::Parameterizer_traits_3<ParameterizationMesh_3> . . . . page 3393
CGAL::Fixed_border_parameterizer_3<ParameterizationMesh_3, BorderParameterizer_3, SparseLinearAlgebraTraits_d> . . . . page 3362
CGAL::Barycentric_mapping_parameterizer_3<ParameterizationMesh_3, BorderParameterizer_3, SparseLinearAlgebraTraits_d> . . . . page 3350
CGAL::Discrete_authalic_parameterizer_3<ParameterizationMesh_3, BorderParameterizer_3, SparseLinearAlgebraTraits_d> . . . . page 3358
CGAL::Discrete_conformal_map_parameterizer_3<ParameterizationMesh_3, BorderParameterizer_3, SparseLinearAlgebraTraits_d> . . . . page 3360
CGAL::LSCM_parameterizer_3<ParameterizationMesh_3, BorderParameterizer_3, SparseLinearAlgebraTraits_d> . . . . page 3365
CGAL::Mean_value_coordinates_parameterizer_3<ParameterizationMesh_3, BorderParameterizer_3, SparseLinearAlgebraTraits_d> . . . . page 3369

See Also

ParameterizationMesh_3 . . . . page 3371

Function

CGAL::parameterize

Definition

CGAL::parameterize() is the main entry-point of the *Surface_mesh_parameterization* package.

It computes a one-to-one mapping from a 3D triangle surface 'mesh' to a simple 2D domain. The mapping is piecewise linear on the triangle mesh. The result is a pair (u,v) of parameter coordinates for each vertex of the input mesh. One-to-one mapping may be guaranteed or not, depending on the chosen *ParametizerTraits* algorithm.

The *CGAL::parameterize()* function exists in two flavors, to provide a default parameterization algorithm of Floater Mean Value Coordinates.

#include <CGAL/parameterize.h>

template<class ParameterizationMesh_3>
Parameterizer_traits_3<ParameterizationMesh_3>::Error_code

parameterize(ParameterizationMesh_3& mesh)

Compute a one-to-one mapping from a 3D triangle surface *mesh* to a 2D circle, using Floater Mean Value Coordinates algorithm. A one-to-one mapping is guaranteed.
The mapping is piecewise linear on the input mesh triangles. The result is a (u, v) pair of parameter coordinates for each vertex of the input mesh.
Preconditions:
mesh must be a surface with one connected component. *mesh* must be a triangular mesh.
Parameters:
mesh: 3D mesh, model of *ParameterizationMesh_3* concept.

template<class ParameterizationMesh_3, class ParameterizerTraits_3>
Parameterizer_traits_3<ParameterizationMesh_3>::Error_code

parameterize(ParameterizationMesh_3& mesh, ParameterizerTraits_3 parameterizer)

Compute a one-to-one mapping from a 3D triangle surface *mesh* to a simple 2D domain. The mapping is piecewise linear on the triangle mesh. The result is a pair (u, v) of parameter coordinates for each vertex of the input mesh.
One-to-one mapping may be guaranteed or not, depending on the chosen *ParametizerTraits_3* algorithm.
Preconditions:
mesh must be a surface with one connected component. *mesh* must be a triangular mesh. The mesh border must be mapped onto a convex polygon (for fixed border parameterizations).
Parameters:
mesh: 3D mesh, model of *ParameterizationMesh_3*. *parameterizer*: Parameterization method for *mesh*.

See Also

CGAL::Barycentric_mapping_parameterizer_3<ParameterizationMesh_3, BorderParameterizer_3, SparseLinearAlgebraTraits_d> . . . . page 3350

Example

See *Simple_parameterization.cpp* example.

Implementation

This function simply calls the parameterize() method of the parameterization algorithm chosen.

Concept

SparseLinearAlgebraTraits_d

Definition

The concept *SparseLinearAlgebraTraits_d* is used to solve sparse linear systems $A*X = B$.

Refines

LinearAlgebraTraits_d

Types

SparseLinearAlgebraTraits_d:: Matrix
SparseLinearAlgebraTraits_d:: Vector
SparseLinearAlgebraTraits_d:: NT

Creation

SparseLinearAlgebraTraits_d sparse_LA; Default constructor.

Operations

bool *sparse_LA.linear_solver(Matrix A, Vector B, Vector& X, NT& D)*

Solve the sparse linear system $A*X = B$. Return true on success. The solution is then (1/D) $*$ X.
Preconditions:
A.*row_dimension*() == B.dimension(). A.*column_dimension*() == X.dimension().

Has Models

CGAL::Eigen_solver_traits<T> . . . . page 3410
OpenNL::DefaultLinearSolverTraits<COEFFTYPE, MATRIX, VECTOR, SOLVER> in OpenNL package
OpenNL::SymmetricLinearSolverTraits<COEFFTYPE, MATRIX, VECTOR, SOLVER> in OpenNL package

See Also

SparseLinearAlgebraTraits_d::Matrix . . . . page 3367
SparseLinearAlgebraTraits_d::Vector . . . . page 3414

CGAL::Square_border_arc_length_parameterizer_3< ParameterizationMesh_3>

Class

Definition

This class parameterizes the border of a 3D surface onto a square, with an arc-length parameterization: (u, v) values are proportional to the length of border edges.

Square_border_parameterizer_3 implements most of the border parameterization algorithm. This class implements only *compute_edge_length*() to compute a segment's length.

#include <CGAL/Square_border_parameterizer_3.h>

Inherits From

Square_border_parameterizer_3<ParameterizationMesh_3>

Is Model for the Concepts

Model of the *BorderParameterizer_3* concept.

Design Pattern

BorderParameterizer_3 models are Strategies [GHJV95]: they implement a strategy of border parameterization for models of *ParameterizationMesh_3*

Parameters

The full template declaration is:

template<class *ParameterizationMesh_3*>
class *Square_border_arc_length_parameterizer_3*;

Creation

Operations

virtual double *bp.compute_edge_length(Adaptor mesh, Vertex_const_handle source, Vertex_const_handle target) [protected, virtual]*

Compute the length of an edge.
Arc-length border parameterization: (u, v) values are proportional to the length of border edges.

See Also

CGAL::Circular_border_arc_length_parameterizer_3<ParameterizationMesh_3> . . . . page 3353
CGAL::Circular_border_uniform_parameterizer_3<ParameterizationMesh_3> . . . . page 3357
CGAL::Square_border_uniform_parameterizer_3<ParameterizationMesh_3> . . . . page 3405
CGAL::Two_vertices_parameterizer_3<ParameterizationMesh_3> . . . . page 3412

Example

See *Square_border_parameterization.cpp* example.

CGAL::Square_border_parameterizer_3<ParameterizationMesh_3>

Class

Definition

This is the base class of strategies that parameterize the border of a 3D surface onto a square. *Square_border_parameterizer_3* is a pure virtual class, thus cannot be instantiated.

It implements most of the algorithm. Subclasses just have to implement *compute_edge_length*() to compute a segment's length.

Implementation note: To simplify the implementation, *BorderParameterizer_3* models know only the *ParameterizationMesh_3* class. They do not know the parameterization algorithm requirements or the kind of sparse linear system used.

#include <CGAL/Square_border_parameterizer_3.h>

Is Model for the Concepts

Model of the *BorderParameterizer_3* concept (although you cannot instantiate this class).

Design Pattern

BorderParameterizer_3 models are Strategies [GHJV95]: they implement a strategy of border parameterization for models of *ParameterizationMesh_3*.

Parameters

The full template declaration is:

template<class *ParameterizationMesh_3*>
class *Square_border_parameterizer_3*;

Types

Square_border_parameterizer_3<ParameterizationMesh_3>:: Adaptor

Export *ParameterizationMesh_3* template parameter.

Creation

Operations

Parameterizer_traits_3<Adaptor>::Error_code

bp.parameterize_border(Adaptor& mesh)

Assign to mesh's border vertices a 2D position (i.e. a (u, v) pair) on border's shape. Mark them as *parameterized*.

bool	*bp.is_border_convex()*	
		Indicate if border's shape is convex.
virtual double	*bp.compute_edge_length(Adaptor mesh, Vertex_const_handle source, Vertex_const_handle target) [protected, pure virtual]*	
		Compute the length of an edge.

See Also

CGAL::Square_border_arc_length_parameterizer_3<ParameterizationMesh_3> . . . . page 3401
CGAL::Square_border_uniform_parameterizer_3<ParameterizationMesh_3> . . . . page 3405

CGAL::Square_border_uniform_parameterizer_3< ParameterizationMesh_3>

Class

Definition

This class parameterizes the border of a 3D surface onto a square in a uniform manner: points are equally spaced.

Square_border_parameterizer_3 implements most of the border parameterization algorithm. This class implements only *compute_edge_length*() to compute a segment's length.

#include <CGAL/Square_border_parameterizer_3.h>

Inherits From

Square_border_parameterizer_3<ParameterizationMesh_3>

Is Model for the Concepts

Model of the *BorderParameterizer_3* concept.

Design Pattern

BorderParameterizer_3 models are Strategies [GHJV95]: they implement a strategy of border parameterization for models of *ParameterizationMesh_3*.

Parameters

The full template declaration is:

template<class *ParameterizationMesh_3*>
class *Square_border_uniform_parameterizer_3*;

Creation

Operations

virtual double *bp.compute_edge_length(Adaptor, Vertex_const_handle, Vertex_const_handle) [protected, virtual]*

Compute the length of an edge.
Uniform border parameterization: points are equally spaced.

See Also

CGAL::Circular_border_arc_length_parameterizer_3<ParameterizationMesh_3> . . . . page 3353
CGAL::Circular_border_uniform_parameterizer_3<ParameterizationMesh_3> . . . . page 3357
CGAL::Square_border_arc_length_parameterizer_3<ParameterizationMesh_3> . . . . page 3401
CGAL::Two_vertices_parameterizer_3<ParameterizationMesh_3> . . . . page 3412

CGAL::Eigen_sparse_matrix<T>

Class

Definition

The class *Eigen_sparse_matrix* is a C++ wrapper around Eigen's matrix type *Eigen::SparseMatrix* that represents general matrices, be they symmetric or not. The version 3.1 (or greater) of EIGEN must be available on the system.

#include <CGAL/Eigen_matrix.h>

Is Model for the Concepts

Model of the *SparseLinearAlgebraTraits_d::Matrix* concept.

Parameters

T: Number type.

Types

Eigen_sparse_matrix<T>:: EigenType	The internal matrix type from EIGEN.

See Also

CGAL::Eigen_solver_traits<T> .... page 3410
CGAL::Eigen_sparse_symmetric_matrix<T> .... page 3408
CGAL::Eigen_vector<T> .... page 3409

Class

CGAL::Eigen_sparse_symmetric_matrix<T>

Definition

The class *Eigen_sparse_symmetric_matrix* is a C++ wrapper around EIGEN's matrix type *Eigen::SparseMatrix*.

As the matrix is symmetric only the lower triangle part is stored.

#include <CGAL/Eigen_matrix.h>

Is Model for the Concepts

Model of the *SparseLinearAlgebraTraits_d::Matrix* concept.

Parameters

T: Number type.

Types

Eigen_sparse_symmetric_matrix<T>:: EigenType

The internal matrix type from EIGEN.

See Also

CGAL::Eigen_solver_traits<T> .... page 3410
CGAL::Eigen_sparse_symmetric_matrix<T> .... page 3408
CGAL::Eigen_vector<T> .... page 3409

CGAL::Eigen_vector<T>

Class

Definition

The class *Eigen_vector* is a C++ wrapper around EIGEN's vector, which is a simple array of numbers. The version 3.1 (or greater) of EIGEN must be available on the system.

#include <CGAL/Eigen_vector.h>

Is Model for the Concepts

SparseLinearAlgebraTraits_d::Vector.

Parameters

T: Number type.

Types

Eigen_vector<T>:: EigenType	The internal vector type from EIGEN.

See Also

CGAL::Eigen_solver_traits<T> . . . . page 3410
CGAL::Eigen_sparse_matrix<T> . . . . page 3407
CGAL::Eigen_sparse_symmetric_matrix<T> . . . . page 3408

CGAL::Eigen_solver_traits<T>

Definition

The class *Eigen_solver_traits* provides an interface to the sparse solvers of EIGEN. The version 3.1 (or greater) of EIGEN must be available on the system.

#include <CGAL/Eigen_solver_traits.h>

Is Model for the Concepts

SparseLinearAlgebraTraits_d.

Parameters

T: a sparse solver of EIGEN. The default solver is the iterative bi-congugate gradient stabilized solver *Eigen::BiCGSTAB* for *double*.

Types

typedef typename T::Scalar

NT;

typedef CGAL::Eigen_vector<NT>

Vector;

Eigen_solver_traits<T>:: Matrix — If *T* is *Eigen::ConjugateGradient<M>* or *Eigen::SimplicialCholesky<M>*, *Matrix* is *CGAL::Eigen_sparse_symmetric_matrix<T>* and *CGAL::Eigen_sparse_matrix<T>* otherwise.

Operations

T& *st.solver()* — Returns a reference to the internal EIGEN solver. This function can be used for example to set specific parameters of the solver.

See Also

CGAL::Eigen_sparse_matrix<T> . . . . page 3407
CGAL::Eigen_sparse_symmetric_matrix<T> . . . . page 3408
CGAL::Eigen_vector<T> . . . . page 3409
`http://eigen.tuxfamily.org`

Example

The instantiation of this class assumes an EIGEN sparse solver is provided. Here are few examples:

```
typedef CGAL::Eigen_sparse_matrix<double>::EigenType EigenMatrix;

//iterative general solver
typedef CGAL::Eigen_solver_traits< Eigen::BiCGSTAB<EigenMatrix> >
    Iterative_general_solver;

//iterative symmetric solver
typedef CGAL::Eigen_solver_traits< Eigen::ConjugateGradient<EigenMatrix>
    > Iterative_symmetric_solver;

//direct symmetric solver
typedef CGAL::Eigen_solver_traits< Eigen::SimplicialCholesky<EigenMatrix>
     > Direct_symmetric_solver;
```

Class

CGAL::Two_vertices_parameterizer_3<ParameterizationMesh_3>

Definition

Two_vertices_parameterizer_3 is the default border parameterizer for Least Squares Conformal Maps parameterization.

The class *Two_vertices_parameterizer_3* parameterizes two extreme vertices of a 3D surface. This kind of border parameterization is used by free border parameterizations.

Implementation note: To simplify the implementation, *BorderParameterizer_3* models know only the *ParameterizationMesh_3* class. They do not know the parameterization algorithm requirements or the kind of sparse linear system used.

#include <CGAL/Two_vertices_parameterizer_3.h>

Is Model for the Concepts

Model of the *BorderParameterizer_3* concept.

Design Pattern

BorderParameterizer_3 models are Strategies [GHJV95]: they implement a strategy of border parameterization for models of *ParameterizationMesh_3*.

Parameters

The full template declaration is:

template<class *ParameterizationMesh_3*>
class *Two_vertices_parameterizer_3*;

Types

Two_vertices_parameterizer_3<ParameterizationMesh_3>:: Adaptor

Export *ParameterizationMesh_3* template parameter.

Creation

Operations

Parameterizer_traits_3<Adaptor>::Error_code

bp.parameterize_border(Adaptor& mesh)

Map two extreme vertices of the 3D mesh and mark them as *parameterized*.

bool *bp.is_border_convex()*

Indicate if border's shape is convex. Meaningless for free border parameterization algorithms.

See Also

CGAL::Circular_border_arc_length_parameterizer_3<ParameterizationMesh_3> . . . . page 3353
CGAL::Circular_border_uniform_parameterizer_3<ParameterizationMesh_3> . . . . page 3357
CGAL::Square_border_arc_length_parameterizer_3<ParameterizationMesh_3> . . . . page 3401
CGAL::Square_border_uniform_parameterizer_3<ParameterizationMesh_3> . . . . page 3405

Concept

SparseLinearAlgebraTraits_d::Vector

Definition

SparseLinearAlgebraTraits_d::Vector is a concept of a vector that can be multiplied by a sparse matrix.

Refines

LinearAlgebraTraits_d::Vector

Types

Vector:: NT

Creation

Vector v(int rows);	Create a vector initialized with zeros.
Vector v(Vector toCopy);	Copy constructor.

Operations

int	*v.dimension() const*	Return the vector's number of coefficients.
NT	*v*[*int row*]	Read/write access to a vector coefficient. *Precondition*: 0 <= row < dimension().
NT&	*v*[*int row*]	

Has Models

Eigen_vector<T>
OpenNL::FullVector<T> in *OpenNL* package

See Also

SparseLinearAlgebraTraits_d . . . . page 3400
SparseLinearAlgebraTraits_d::Matrix . . . . page 3367

Chapter 55

2D Placement of Streamlines

Abdelkrim Mebarki

Contents

This chapter describes the CGAL 2D streamline placement package. Basic definitions and notions are given in Section 55.1. Section 55.2 gives a description of the integration process. Section 55.3 provides a brief description of the algorithm. Section 55.4 presents the implementation of the package, and Section 55.5 details two example placements.

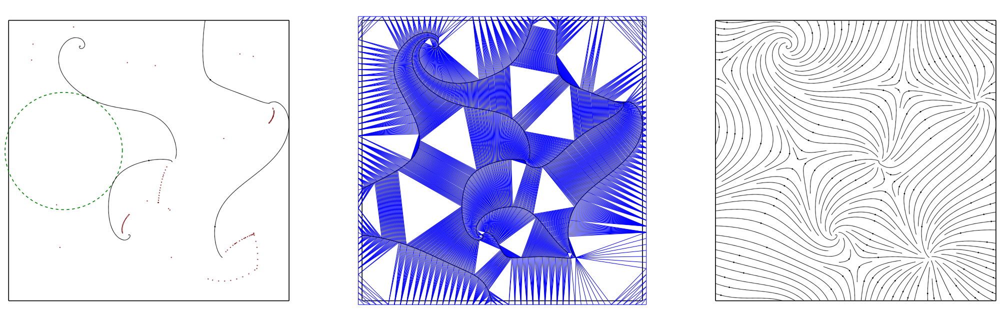


Figure 55.1: The core idea of the algorithm is to integrate the streamlines from the center of the biggest empty cavities in the domain (left). A Delaunay triangulation of all the sample points is used to model the streamlines and the spaces within the domain (middle). A final result is shown (right).

55.1 Definitions

In physics, a *field* is an assignment of a quantity to every point in space. For example, a gravitational field assigns a gravitational potential to each point in space.

Vector and direction fields are commonly used for modeling physical phenomena, where a direction and magnitude, namely a vector is assigned to each point inside a domain (such as the magnitude and direction of the force at each point in a magnetic field).

Streamlines are important tools for visualizing flow fields. A *streamline* is a curve everywhere tangent to the field. In practice, a streamline is often represented as a polyline (series of points) iteratively elongated by bidirectional numerical integration started from a *seed point*, until it comes close to another streamline (according to a specified distance called *the separating distance*), hits the domain boundary, reaches a critical point or generates a closed path.

A *valid* placement of streamlines consists of saturating the domain with a set of tangential streamlines in accordance with a specified *density*, determined by the *separating distance* between the streamlines.

55.2 Fundamental Notions

A streamline can be considered as the path traced by an imaginary massless particle dropped into a steady fluid flow described by the field. The construction of this path consists in the solving an ordinary differential equation for successive time intervals. In this way, we obtain a series of points $p_k, 0 < k < n$ which allow visualizing the streamline. The differential equation is defined as follows :

$$\frac{dp}{dt} = v(p(t)), \qquad p(0) = p_0$$

where *p(t)* is the position of the particle at time t, v is a function which assigns a vector value at each point in the domain (possibly by interpolation), and p_0 is the initial position. The position after a given interval Δt is given by :

$$p(t+\Delta t) = p(t) + \int_t^{t+\Delta t} v(p(t))dt$$

. Several numeric methods have been proposed to solve this equation. In this package, the Euler, and the Second Order Runge-Kutta algorithm are implemented.

55.2.1 Euler Integrator

This algorithm approximates the point computation by this formula

$$p_{k+1} = p_k + hv(p_k)$$

where h specifies the *integration step* (see Figure 55.2). The integration can be done forward (resp. backward) by specifying a positive (resp. negative) integration step h. The streamline is then constructed by successive integration from a seed point both forward and backward.

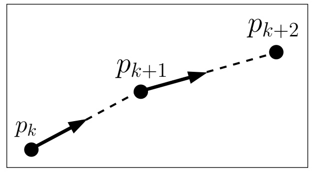


Figure 55.2: Euler integrator.

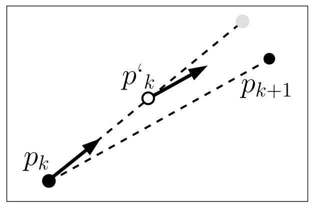


Figure 55.3: Runge-Kutta second order integrator (The empty circle represents the intermediate point, and the gray disk represents the Euler integrated point).

55.2.2 Second Order Runge-Kutta Integrator

This method introduces an intermediate point p'_k between p_k and p_{k+1} to increase the precision of the computation (see Figure 55.3), where:

$$\begin{array}{rcccl} p'_k & = & p_k & + & \frac{1}{2}hv(p_k) \\ p_{k+1} & = & p_k & + & hv(p'_k) \end{array}$$

See [PTVF02] for further details about numerical integration.

55.3 Farthest Point Seeding Strategy

The algorithm implemented in this package [MAD05] consists of placing one streamline at a time by numerical integration starting farthest away from all previously placed streamlines.

The input of our algorithm is given by (i) a flow field, (ii) a *density* specified either globally, by the inverse of the ideal spacing distance, or locally by a density field, and (iii) a *saturation* ratio over the desired spacing required to trigger the seeding of a new streamline.

The input flow field is given by a discrete set of vectors or directions sampled within a domain, associated with an interpolation scheme (*e.g.* bilinear interpolation over a regular grid, or natural neighbor interpolation over an irregular point set to allow for an evaluation at each point coordinate within the domain).

The *output* is a streamline placement, represented as a list of streamlines. The core idea of our algorithm consists of placing one streamline at a time by numerical integration seeded at the farthest point from all previously placed streamlines.

The streamlines are approximated by polylines, whose points are inserted to a 2D Delaunay triangulation (see figure 55). The empty circumscribed circles of the Delaunay triangles provide us with a good approximation of the cavities in the domain.

After each streamline integration, all incident triangles whose circumcircle diameter is larger (within the saturation ratio) than the desired spacing distance are pushed to a priority queue sorted by the triangle circumcircle diameter. To start each new streamline integration, the triangle with largest circumcircle diameter (and hence the biggest cavity) is popped out of the queue. We first test if it is still a valid triangle of the triangulation, since it could have been destroyed by a streamline previously added to the triangulation. If it is not, we pop another triangle out of the queue. If it is, we use the center of its circumcircle as seed point to integrate a new streamline.

Our algorithm terminates when the priority queue is empty. The size of the biggest cavity being monotonically decreasing, our algorithm guarantees the domain saturation.

55.4 Implementation

Streamlines are represented as polylines, and are obtained by iterative integration from the seed point. A polyline is represented as a range of points. The computation is processed via a list of Delaunay triangulation vertices.

To implement the triangular grid, the CGAL *Delaunay_triangulation_2* class is used. The priority queue used to store candidate seed points is taken from the Standard Template Library [Sil97].

55.5 Examples

The first example illustrates the generation of a 2D streamline placement from a vector field defined on a regular grid.

```
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/Stream_lines_2.h>
#include <CGAL/Runge_kutta_integrator_2.h>
#include <CGAL/Regular_grid_2.h>
#include <CGAL/Triangular_field_2.h>

#include <iostream>
#include <fstream>

typedef CGAL::Exact_predicates_inexact_constructions_kernel         K;
typedef CGAL::Regular_grid_2<K> Field;
typedef CGAL::Runge_kutta_integrator_2<Field> Runge_kutta_integrator;
typedef CGAL::Stream_lines_2<Field, Runge_kutta_integrator> Strl;
typedef Strl::Point_iterator_2 Point_iterator;
```

```
typedef Strl::Stream_line_iterator_2 Strl_iterator;
typedef Strl::Point_2 Point_2;
typedef Strl::Vector_2 Vector_2;

int main()
{
  Runge_kutta_integrator runge_kutta_integrator;

  /*data.vec.cin is an ASCII file containing the vector field values*/
  std::ifstream infile("data/vnoise.vec.cin", std::ios::in);
  double iXSize, iYSize;
  unsigned int x_samples, y_samples;
  iXSize = iYSize = 512;
  infile >> x_samples;
  infile >> y_samples;
  Field regular_grid_2(x_samples, y_samples, iXSize, iYSize);
  /*fill the grid with the appropreate values*/
  for (unsigned int i=0;i<x_samples;i++)
    for (unsigned int j=0;j<y_samples;j++)
      {
        double xval, yval;
        infile >> xval;
        infile >> yval;
        regular_grid_2.set_field(i, j, Vector_2(xval, yval));
      }
  infile.close();

  /* the placement of streamlines */
  std::cout << "processing...\n";
  double dSep = 3.5;
  double dRat = 1.6;
  Strl Stream_lines(regular_grid_2, runge_kutta_integrator,dSep,dRat);
  std::cout << "placement generated\n";

  /*writing streamlines to streamlines_on_regular_grid_1.stl */
  std::ofstream fw("streamlines_on_regular_grid_1.stl",std::ios::out);
  fw << Stream_lines.number_of_lines() << "\n";
  for(Strl_iterator sit = Stream_lines.begin(); sit !=
   Stream_lines.end(); sit++)
    {
      fw << "\n";
      for(Point_iterator pit = sit->first; pit != sit->second; pit++){
        Point_2 p = *pit;
        fw << p.x() << " " << p.y() << "\n";
      }
    }

  fw.close();

}
```

File: examples/Stream_lines_2/stl_regular_field.cpp

The second example depicts the generation of a streamline placement from a vector field defined on a triangular grid.

```
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/Stream_lines_2.h>
#include <CGAL/Runge_kutta_integrator_2.h>
#include <CGAL/Triangular_field_2.h>

#include <iostream>
#include <fstream>

typedef CGAL::Exact_predicates_inexact_constructions_kernel         K;
typedef K::Point_2
    Point;
typedef K::Vector_2
    Vector;
typedef CGAL::Triangular_field_2<K>
    Field;
typedef CGAL::Runge_kutta_integrator_2<Field>
    Runge_kutta_integrator;
typedef CGAL::Stream_lines_2<Field, Runge_kutta_integrator>         Strl;
typedef Strl::Stream_line_iterator_2
    stl_iterator;

int main()
{
  Runge_kutta_integrator runge_kutta_integrator(1);

  /*datap.tri.cin and datav.tri.cin are ascii files where are stored the
    vector velues*/
  std::ifstream inp("data/datap.tri.cin");
  std::ifstream inv("data/datav.tri.cin");
  std::istream_iterator<Point> beginp(inp);
  std::istream_iterator<Vector> beginv(inv);
  std::istream_iterator<Point> endp;

  Field triangular_field(beginp, endp, beginv);

  /* the placement of streamlines */
  std::cout << "processing...\n";
  double dSep = 30.0;
  double dRat = 1.6;
  Strl Stream_lines(triangular_field, runge_kutta_integrator,dSep,dRat);
  std::cout << "placement generated\n";

  /*writing streamlines to streamlines.stl */
  std::cout << "streamlines.stl\n";
  std::ofstream fw("streamlines.stl",std::ios::out);
  Stream_lines.print_stream_lines(fw);
}
```

File: examples/Stream_lines_2/stl_triangular_field.cpp

2D Placement of Streamlines Reference Manual

Abdelkrim Mebarki

Vector and direction fields are commonly used for modeling physical phenomena, where a direction and magnitude, namely a vector is assigned to each point inside a domain.

A streamline is a curve everywhere tangent to the field. It can be considered as the path traced by an imaginary massless particle dropped into a steady fluid flow described by the field.

A streamline is represented as a polyline iteratively elongated by bidirectional numerical integration started from a seed point, until it comes close to another streamline, hits the domain boundary, or reaches a critical point.

The *Stream_lines_2* class consists of saturating the domain with a set of tangential streamlines in accordance with a specified density.

Streamlines are represented as containers of points, manipulated by an iterator range of points, and the whole placement is accessible via an iterator range of streamlines.

The main class in this package, the *Stream_lines_2* class of CGAL depends on two template parameters. The first template parameter stands for a class which represents both the vector field and the visualization domain with operations on them, and should be instantiated by a model of the concept *VectorField_2*. The second template parameter stands for a function object that ensures the numerical integration used to construct the streamlines, and should be instantiated by a model of the concept *Integrator_2*.

55.6 Classified Reference Pages

Concepts

Classes

55.7 Alphabetical List of Reference Pages

CGAL::Euler_integrator_2<VectorField_2>

Definition

This class implements the first order Euler integrator. The template parameter *VectorField_2* has to be instantiated by a model of the concept *VectorField_2*.

Creation

Euler_integrator_2<VectorField_2> einteg(FT integration_step);

Creates an Euler integrator class *einteg* with *integration_step* as integration step.

Is Model for the Concepts

Integrator_2

See Also

Runge_kutta_integrator_2<VectorField_2>

Functor Concept

Integrator_2

Definition

The concept Integrator_2 describes the set of requirements to be fulfilled by any function object used to instantiate the second template parameter of the class *Stream_lines_2<VectorField_2,Integrator_2>*. This concept provides the operation that integrates a new point from a given point with a predefined step, and according to a specified vector.

Types

Integrator_2:: FT	The scalar type.
Integrator_2:: Point_2	The point type.
Integrator_2:: Vector_2	The vector type.
Integrator_2:: Vector_field_2	The vector field type.

Creation

Integrator_2 integ;	only a default constructor is needed.

Operations

The following operations return the newly integrated point.

Point_2 *integ(Point_2 p, Vector_field_2 vector_field_2)*

returns the new position from the actual position defined by *p*, according to the vector given by *vector_field_2* at *p*.
Precondition: *vector_field_2.is_in_domain(p)* must be true.

Point_2 *integ(Point_2 p, Vector_field_2 vector_field_2, FT integration_step)*

As above. The integration step is defined by *integration_step*.
Precondition: *vector_field_2.is_in_domain(p)* must be true.

Point_2 *integ(Point_2 p, Vector_field_2 vector_field_2, FT integration_step, bool direction)*

As above. In addition, this function integrates forward if *direction* is true, and backward if it is false.
Precondition: *vector_field_2.is_in_domain(p)* must be true.

Has Models

CGAL::Euler_integrator_2<VectorField_2>
CGAL::Runge_kutta_integrator_2<VectorField_2>

CGAL::Regular_grid_2<StreamLinesTraits_2>

Class

Definition

The template parameter *StreamLinesTraits_2* has to be instantiated by a model of the concept *StreamLinesTraits_2*.

This class provides a 2D vector field specified by a set of sample points defined on a regular grid, with a bilinear interpolation scheme over its cells (i.e. for each point p in a cell c, the vector value is interpolated from the vertices of c).

Types

typedef StreamLinesTraits_2::FT	*FT;*	the scalar type.
typedef StreamLinesTraits_2::Point_2	*Point_2;*	the point type.
typedef StreamLinesTraits_2::Vector_2	*Vector_2;*	the vector type.

Creation

Regular_grid_2<StreamLinesTraits_2> rgrid(int x_samples, int y_samples, FT x_size, FT y_size);

Generate a regular grid *rgrid* whose size is *x_size* by *y_size*, while *x_samples* and *y_samples* specify the number of samples on x and y.

Modifiers

In addition to the minimum interface required by the concept definition, the class *Regular_grid_2< StreamLinesTraits_2>* provides the following function to fill the vector field with the user data.

void *rgrid.set_xy(int i, int j, Vector_2 v)*

Attribute the vector v to the position (i,j) on the regular grid.

Access Functions

std::pair<int, int>	*rgrid.get_dimension()*	returns the dimension of the grid.
std::pair<FT, FT>	*rgrid.get_size()*	returns the size of the grid.

Is Model for the Concepts

VectorField_2

See Also

Triangular_field_2<StreamLinesTraits_2>

CGAL::Runge_kutta_integrator_2<VectorField_2>

Functor
Class

Definition

The template parameter *VectorField_2* has to be instantiated by a model of the concept *VectorField_2*. This class implements the second order Runge-Kutta integrator.

Creation

Runge_kutta_integrator_2<VectorField_2> rkinteg(FT integration_step);

Creates a Runge-Kutta second order integrator class *rkinteg* with *integration_step* as integration step.

Is Model for the Concepts

Integrator_2

See Also

Euler_integrator_2<VectorField_2>

StreamLinesTraits_2

Definition

The concept StreamLinesTraits_2 describes the set of requirements to be fulfilled by any class used to instantiate the template parameter of the class *Regular_grid_2<StreamLinesTraits_2>*. This concept provides the types handled by the *Stream_lines_2<VectorField_2, Integrator_2>* class.

Types

StreamLinesTraits_2:: FT	The scalar type.
StreamLinesTraits_2:: Point_2	The point type.
StreamLinesTraits_2:: Vector_2	The vector type.

Has Models

The kernels of CGAL are models for this traits class.

CGAL::Stream_lines_2<VectorField_2,Integrator_2>

Class

Definition

The class *Stream_lines_2<VectorField_2,Integrator_2>* is designed to handle a placement of streamlines in a 2D domain according to a bidimensional vector field.

The class *Stream_lines_2<VectorField_2,Integrator_2>* creates a placement of streamlines according to a specified density and gives access to those streamlines via two iterators over a container of iterators that provide access to the streamline points.

Parameters

The class *Stream_lines_2<VectorField_2,Integrator_2>* has two template parameters. The first parameter *VectorField_2* has to be instantiated by a model of the concept *VectorField_2*. The second parameter is the function object *Integrator_2*, and has to be instantiated by a model of the concept *Integrator_2*.

Types

typedef VectorField_2::Geom_traits	*Geom_traits;*	the traits class.
typedef VectorField_2::FT	*FT;*	the scalar type.
typedef VectorField_2::Point_2	*Point_2;*	the point type.
typedef VectorField_2::Vector_2	*Vector_2;*	the vector type.

The class *Stream_lines_2<VectorField_2,Integrator_2>* provides also two types for handling streamlines:

Stream_lines_2<VectorField_2,Integrator_2>:: Point_iterator_2

iterator of points with value type *Point_2*.

Stream_lines_2<VectorField_2,Integrator_2>:: Stream_line_iterator_2

an iterator to visit the streamlines with value type *std::pair< Point_iterator_2, Point_iterator_2>*.

Creation

Stream_lines_2<VectorField_2,Integrator_2> stl(VectorField_2 vector_field_2,
Integrator_2 integrator_2,
FT separating_distance,
FT saturation_ratio)

Generates a streamline placement *stl*.

Modifiers

void *stl.set_separating_distance(FT new_value)*

Modify the separating distance.

void	*stl.set_saturation_ratio(FT new_value)*	
		Modify the saturation ratio.
void	*stl.update()*	Update the placement after changing the separating distance or the saturation ratio.

Access Functions

void	*stl.get_separating_distance()*	returns the separating distance.
void	*stl.get_saturation_ratio()*	returns the saturation ratio.
void	*stl.print_stream_lines(std::ofstream & fw)*	
		prints the streamlines to an ASCII file : line by line, and point by point.

Streamline iterators

The following iterators allow to visit all the streamlines generated by the constructor or the update function.

Stream_line_iterator	*stl.begin() const*	Starts at the first streamline
Stream_line_iterator	*stl.end() const*	Past-the-end iterator

CGAL::Triangular_field_2<StreamLinesTraits_2>

Definition

The template parameter *StreamLinesTraits_2* has to be instantiated by a model of the concept *StreamLinesTraits_2*.

This class provides a vector field specified by a set of sample points defined on a triangulated domain. All sample points are inserted to a *Delaunay triangulation*, and for each point p in the domain located in a face f, its vector value is interpolated from the vertices of the face f.

Types

typedef StreamLinesTraits_2::FT	*FT;*	the scalar type.
typedef StreamLinesTraits_2::Point_2	*Point_2;*	the point type.
typedef StreamLinesTraits_2::Vector_2	*Vector_2;*	the vector type.

Creation

Triangular_field_2<StreamLinesTraits_2> tfield_2(InputIterator1 first_point, InputIterator1 last_point, InputIterator2 first_vector)

Defines the points in the range [*first_point*, *last_point*) as the sample points of the grid, with the corresponding number of vectors started at *first_vector*.
Precondition: The *value_type* of *InputIterator1* is *Point*.
Precondition: The *value_type* of *InputIterator2* is *Vector*.

Is Model for the Concepts

VectorField_2

See Also

Regular_grid_2<StreamLinesTraits_2>

Concept

VectorField_2

Definition

The concept VectorField_2 describes the set of requirements to be fulfilled by any class used to instantiate the first template parameter of the class *Stream_lines_2<VectorField_2,Integrator_2>*. This concept provides the types of the geometric primitives used in the placement of streamlines and some functions for answering different queries.

Types

VectorField_2:: Geom_traits	The traits class.
VectorField_2:: FT	The scalar type.
VectorField_2:: Point_2	The point type.
VectorField_2:: Vector_2	The vector type.

Creation

VectorField_2 vfield;

Any constructor has to allow the user to fill the vector values (i.e. assign a vector to each position within the domain).

Query Functions

Geom_traits::Iso_rectangle_2 *vfield.bbox()*

returns the bounding box of the whole domain.

std::pair<Vector_2,FT> *vfield.get_field(Point_2 p)*

returns the vector field value and the local density.
Precondition: *is_in_domain(p)* must be true.

bool *vfield.is_in_domain(Point_2 p)*

returns true if the point p is inside the domain boundaries, false otherwise.

FT *vfield.get_integration_step(Point_2 p)*

returns the integration step at the point p (i.e. the distance between *p* and the next point in the polyline.).
Precondition: *is_in_domain(p)* must be true.

Has Models

CGAL::Regular_grid_2<StreamLinesTraits_2>
CGAL::Triangular_field_2<StreamLinesTraits_2>

Chapter 56

Approximation of Ridges and Umbilics on Triangulated Surface Meshes

Marc Pouget and Frédéric Cazals

Contents

This chapter describes the CGAL package for the approximating the ridges and umbilics of a smooth surface discretized by a triangle mesh. Given a smooth surface, a ridge is a curve along which one of the principal curvatures has an extremum along its curvature line. An umbilic is a point at which both principal curvatures are equal. Ridges define a singular curve, i.e., a self-intersecting curve, and umbilics are special points on this curve. Ridges are curves of *extremal* curvature and therefore encode important informations used in segmentation, registration, matching and surface analysis. Based on the results of the article [CP05c], we propose algorithms to identify and extract different parts of this singular ridge curve as well as umbilics on a surface given as a triangulated surface mesh. Differential quantities associated to the mesh vertices are assumed to be given for these algorithms; such quantities may be computed by the package *Estimation of Local Differential Properties of Sampled Surfaces via Polynomial Fitting*.

Note that this package needs the third party library Eigen (Page 28)for linear algebra operations.

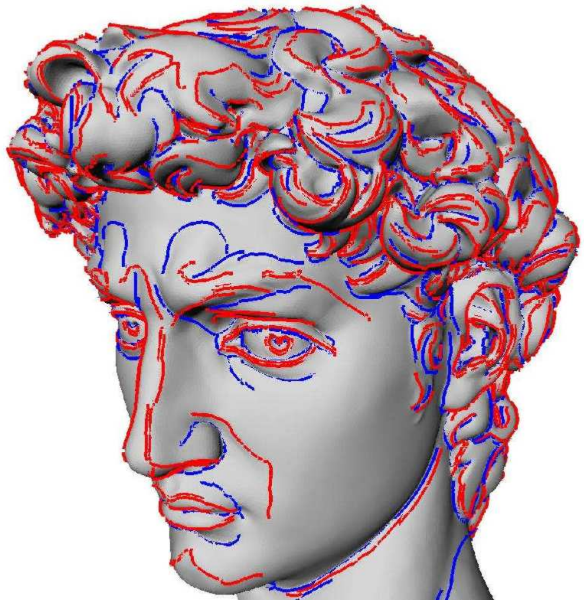


Figure 56.1: Crest ridges on the David, model provided by the Digital Michelangelo Project.

56.0.1 Overview

Section 56.1 presents the basics of the theory of ridges and umbilics on smooth surfaces. Sections 56.2 and 56.3 present algorithms for approximating the ridges and umbilics (of a smooth surface) from a triangle mesh. Section 56.4 gives the package specifications, while example calls to functions of the package are provided in Section 56.5.

56.1 Ridges and Umbilics of a Smooth Surface

For a detailed introduction to ridges and related topics, the reader may consult [HGY+99, Por01], as well as the following survey article [CP05b]. In the sequel, we just introduce the basic notions so as to explain our algorithms. Consider a smooth embedded surface, and denote k_1 and k_2 the principal curvatures, with $k_1 \geq k_2$. Umbilics are the points where $k_1 = k_2$. For any non umbilical point, the corresponding principal directions of curvature are well defined, and we denote them d_1 and d_2. In local coordinates, we denote $\langle , \rangle$ the inner product induced by the ambient Euclidean space, and dk_1, dk_2 the gradients of the principal curvatures. Ridges, illustrated in Figure 56.2 for the standard ellipsoid, are defined by:

Definition. 1 *A non umbilical point is called*

- *a max ridge point, if the* extremality coefficient $b_0 = \langle dk_1, d_1 \rangle$ *vanishes, i.e.* $b_0 = 0$.
- *a min ridge point, if the* extremality coefficient $b_3 = \langle dk_2, d_2 \rangle$ *vanishes, i.e.* $b_3 = 0$ [1].

The previous characterization of ridges involves third-order differential properties. Using fourth-order differential quantities, a ridge point can further be qualified as *elliptic* if it corresponds to a maximum of k_1 or a minimum of k_2, or *hyperbolic* otherwise. Hence we end up with four types of ridges, namely: *MAX_ELLIPTIC_RIDGE*,

[1] Notations b_0, b_3 comes from Equation 57.2

MAX_HYPERBOLIC_RIDGE, *MIN_ELLIPTIC_RIDGE*, *MIN_HYPERBOLIC_RIDGE*, which are illustrated in Figure 56.2. Also of interest are the *crest lines*, a crest line being an elliptic ridge which is a maximum of $\max(|k_1|, |k_2|)$. Crest lines form a subset of elliptic ridges, and can be seen as the visually most salient curves on a surface. Hence we provide the two additional ridge types *MAX_CREST_RIDGE* and *MIN_CREST_RIDGE*, which are illustrated in Figure 56.

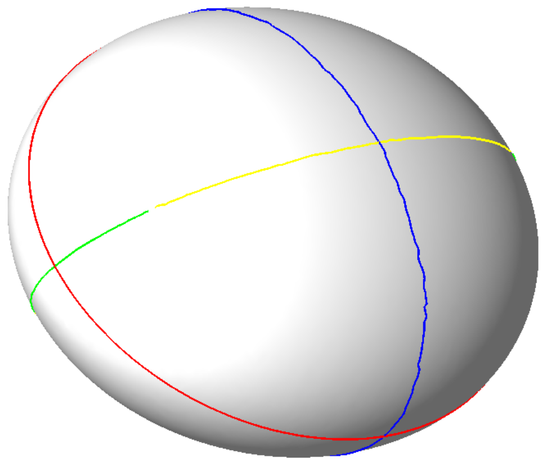


Figure 56.2: Ridges on the ellipsoid, normals pointing outward. Color coding: *MAX_ELLIPTIC_RIDGE* are blue, *MAX_HYPERBOLIC_RIDGE* are green, *MIN_ELLIPTIC_RIDGE* are red and *MIN_HYPERBOLIC_RIDGE* are yellow. The red line is also the *MIN_CREST_RIDGE* and this is the only crest ridge of the ellipsoid.

At any point of the surface which is not an umbilic, principal directions d_1, d_2 are well defined, and these (non oriented) directions together with the normal vector n define two direct orthonormal frames. If v_1 is a unit vector of direction d_1 then there exists a unique unit vector v_2 so that (v_1, v_2, n) is direct, that is has the same orientation as the canonical basis of the ambient $3d$ space (and the other possible frame is $(-v_1, -v_2, n)$). In the coordinate systems (v_1, v_2, n), the surface has the following canonical form, known as the Monge form :

$$z(x,y) = \frac{1}{2}(k_1x^2 + k_2y^2) + \frac{1}{6}(b_0x^3 + 3b_1x^2y + 3b_2xy^2 + b_3y^3) \tag{56.1}$$

$$+\frac{1}{24}(c_0x^4 + 4c_1x^3y + 6c_2x^2y^2 + 4c_3xy^3 + c_4y^4) + h.o.t \tag{56.2}$$

The Taylor expansion of k_1 (resp. k_2) along the max (resp. min) curvature line going through the origin and parameterized by x (resp. y) are:

$$k_1(x) = k_1 + b_0x + \frac{P_1}{2(k_1 - k_2)}x^2 + h.o.t, \qquad P_1 = 3b_1^2 + (k_1 - k_2)(c_0 - 3k_1^3). \tag{56.3}$$

$$k_2(y) = k_2 + b_3y + \frac{P_2}{2(k_2 - k_1)}y^2 + h.o.t, \qquad P_2 = 3b_2^2 + (k_2 - k_1)(c_4 - 3k_2^3). \tag{56.4}$$

Notice also that switching from one to the other of the two afore-mentioned coordinate systems reverts the sign of all the odd coefficients on the Monge form of the surface.

Hence ridge types are characterized by

- max ridge if $b_0 = 0$
- *MAX_ELLIPTIC_RIDGE* if $b_0 = 0$ and $P_1 < 0$
- *MAX_HYPERBOLIC_RIDGE* if $b_0 = 0$ and $P_1 > 0$

- min ridge if $b_3 = 0$
- *MIN_ELLIPTIC_RIDGE* if $b_3 = 0$ and $P_2 < 0$
- *MIN_HYPERBOLIC_RIDGE* if $b_3 = 0$ and $P_2 > 0$
- *MAX_CREST_RIDGE* if $b_0 = 0$ and $P_1 < 0$ and $|k_1| > |k_2|$
- *MIN_CREST_RIDGE* if $b_3 = 0$ and $P_2 < 0$ and $|k_2| > |k_1|$

As illustrated in Figures 56.3 and 56.4, the patterns made by curvature lines around an umbilic can be characterized using the notion of an *index* associated to the principal directions — see also [CP05b]. As depicted in Figure 56.3, consider a small circuit C around the umbilic, and a point $p \in C$. Starting from an initial orientation u of a tangent vector to the curvature line through point p, propagate *by continuity* this orientation around the circuit. The index is defined by the angle swept by u around this revolution, normalized by 2π. In our example, the index is thus 1/2.

If the index of the principal direction field is $1/2$ then it is called a *ELLIPTIC_UMBILIC*, if it is $-1/2$ it is called a *HYPERBOLIC_UMBILIC*. Otherwise the umbilic is qualified *NON_GENERIC_UMBILIC*.

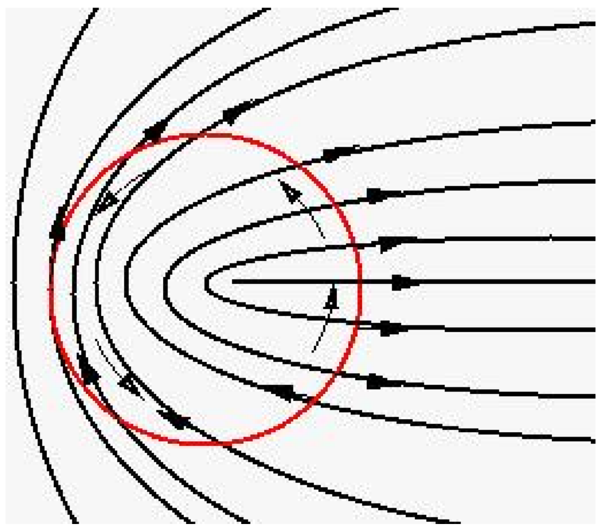


Figure 56.3: Index $1/2$ umbilic or elliptic umbilic.

56.2 Approximating Ridges on Triangulated Surface Meshes

Our method aims at reporting ridges as polygonal lines living on the mesh. It assumes differential quantities are available for each vertex of the mesh (principal curvatures and directions together with third order quantities b_0, b_3 and optionally fourth order quantities P_1, P_2). These differential quantities may be computed for the smooth surface the mesh is inscribed in (analytically or using approximation methods), or may be estimated for a mesh given without reference to an underlying smooth surface. Although the ridge approximation algorithm is the same in both cases, one cannot ambition to ask for the same certificates. This distinction calls for the notion of *compliant* mesh.

Compliant meshes. Ridges of a smooth surface are points with prescribed differential properties, and reporting them from a mesh inscribed in the surface requires delicate hypothesis on the geometry of that mesh so as

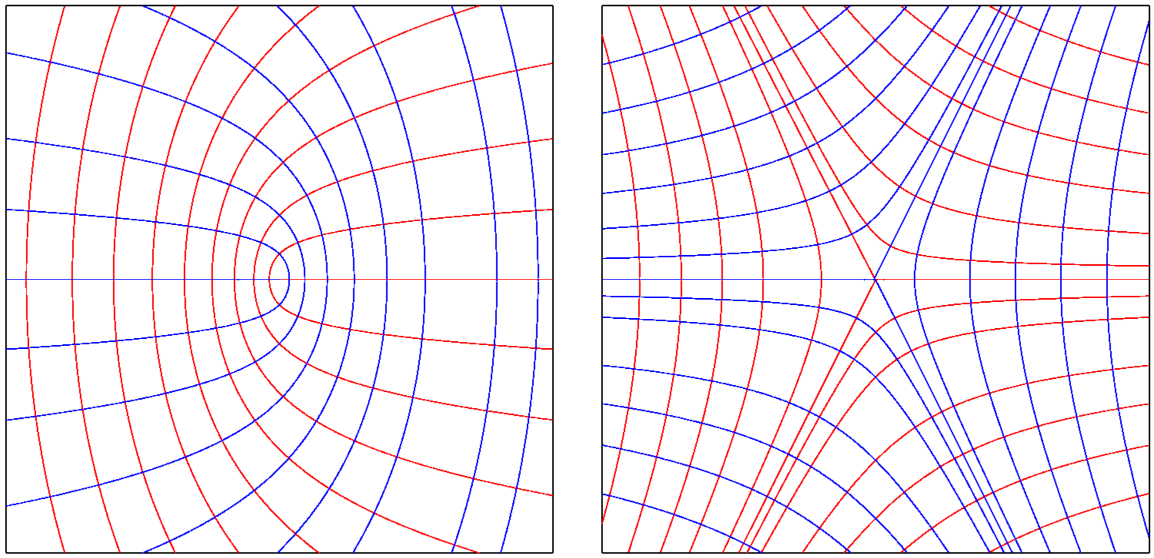


Figure 56.4: Elliptic and hyperbolic umbilics.

to get a certified result. In this paragraph, we assume the mesh provided complies with a number of hypothesis, which guarantee the topology of the ridges reported matches that of the ridges on the smooth surface. To summarize things, a compliant mesh is a mesh dense enough so that (i) umbilics are properly isolated (ii) ridges running next to one another are also properly separated. See [CP05c] for a detailed discussion of *compliant* meshes.

As 0-level set of the extremality coefficients b_0 and b_3, ridges are extracted by a marching triangles algorithm.[2]

As the signs of these extremality coefficients depend on the orientation of the principal directions, we expect both extremalities and vectors orienting the principal direction to be given at each point vertex of the mesh. Except in the neighborhood of umbilics, if the mesh is dense enough, a coherent orientation of principal directions at both endpoints of an edge is chosen such that the angle between the two vectors is acute. This rule, the *acute rule*, is precisely analyzed in [CP05c]. Moreover, we only seek ridges in triangles for which one can find an orientation of its three vertices such that the three edges are coherently oriented by the acute rule. Such triangles are called *regular*. This said, two remarks are in order.

—*Regular triangles and ridge segments.* A regular triangle has 0 or 2 edges crossed by a max (resp. min) ridge, which is tantamount to a sign change of b_0 (resp. b_3) along the corresponding edges. In the latter case, we say that the triangle contains a ridge segment. Two methods are provided to compute its type, be it elliptic or hyperbolic. First, if fourth order differential quantities are provided, one can use the P_1 (P_2) values of Equations 56.3 (56.4) for a max (min) ridge. The value of P_i for a ridge segment is defined as the mean value of the P_i values of the two crossing points on edges (while the value at a crossing point on an edge is the b_i-weighted mean value of the values at endpoints). Alternatively, if third order differential quantities only are available, one may use the geometric method developed in [CP05c].

Using the notion of ridge segment, a *Ridge_line* is defined as a maximal connected sequence of ridge segments of the same type and connected together. Notice that the topology of a *Ridge_line* is either that of an interval or a circle.

—*Non-regular triangles.* In the neighborhood of umbilics, triangle are less likely to be regular and the detec-

[2]A marching triangles algorithm is similar to a 2d marching cubes algorithm (or marching rectangles algorithm), except that a one-manifold is reported on a two-manifold tessellated by triangles.

tion of ridges cannot be relevant by this method. This is why we propose another method to detect umbilics independently.

Non compliant meshes: filtering ridges on *strength* and *sharpness*. For real world applications dealing with coarse meshes, or meshes featuring degenerate regions or sharp features, or meshes conveying some amount of noise, the *compliance* hypothesis [CP05c] cannot be met. In that case, it still makes sense to seek loci of points featuring extremality of estimated principal curvatures, but the ridges reported may require filtering. For example, if the principal curvatures are constant —which is the case on a plane or a cylinder, then all points are ridge points. In this context, an appealing notion is that of *sharp* ridge or *prominent* ridge. Since ridges are witnessed by zero crossings of b_0 and b_3, one can expect erroneous detections as long as these coefficients remain small. In order to select the most prominent ridge points, we focus on points where the variation of the curvature is fast along the curvature line. One can observe that, at a ridge point, according to Equation 56.3, the second derivative of k_1 along its curvature line satisfies $k_1^{''}(0) = P_1/(k_1 - k_2)$. Using this observation, one can define the *sharpness of a ridge* as the integral of the absolute value of $P_1/(k_1 - k_2)$ along the ridge. As the second derivative of the curvature is homogeneous to the inverse of the cube of a length, the sharpness is homogeneous to the inverse of the square of a length. Multiplying the sharpness by the square of the model size gives a threshold and an associated sharpness-filter which are scale independent. Another filtering is also available with the *strength* which is the integral of the curvature along the ridge line [OBS04].

56.3 Approximating Umbilics on Triangulated Surface Meshes

The method aims at identifying some vertices of a mesh as umbilics. It assumes principal curvatures and directions are given at each vertex on the mesh.

Algorithm. Assume each vertex v of the mesh comes with a patch (a topological disk) around it. Checking whether vertex v is an umbilic is a two stages process, which are respectively concerned with the variation of the function $k_1 - k_2$ over the patch, and the index of the vertex computed from the boundary of the patch. More precisely, vertex v is declared to be an umbilic if the following two conditions are met:

- the function $k_1 - k_2$ has its minimum at v amongst all the vertices of the patch;
- the deviation δ of any principal direction along the patch boundary, traversed counter-clock-wise (CCW), has prescribed properties:
  - $\delta \in]\pi/2, 3\pi/2[$, then the umbilic is called elliptic,
  - $\delta \in]-3\pi/2, -\pi/2[$, then the umbilic is called a hyperbolic,
  - otherwise the umbilic is called non-generic.

Finding patches around vertices. Given a vertex v and a parameter t, we aim at defining a collection of triangles around v so that (i) this collection defines a topological disk on the triangulation T and (ii) its size depends on t. First we collect the 1-ring triangles. We define the size s of this 1-ring patch as the (Euclidean) distance from v to its farthest 1-ring vertex neighbor. We then collect recursively adjacent triangles so that the patch remains a topological disk and such that these triangles are at distance less than $s \times t$. Parameter t is the only parameter of the algorithm.

Umbilical patches versus ridges. On a generic surface, generic umbilics are traversed by one or three ridges. For compliant meshes, an umbilic can thus be connected to the ridge points located on the boundary of its patch. This functionality is not provided, and the interested reader is referred to [CP05c] for more details.

56.4 Software Design

All classes of this package are templated by the parameter *TriangulatedSurfaceMesh*, which defines the type of the mesh to which the approximation algorithms operate.

The differential quantities are provided at vertices of this mesh via property maps, a concept commonly used in the Boost library. Scalar data (curvatures and their derivatives) are provided via *Vertex2FTPropertyMap* concepts, while 3*d* vector data (principal directions of curvatures) are provided via *Vertex2VectorPropertyMap* concepts. The rationale for introducing these concepts is that properties are used independently from the way they are stored. This enables the user to store them *internally* in extended vertices or *externally* with maps. We provide a class *Vertex2Data_Property_Map_with_std_map* to adapt *std::map* with a *boost::associative_property_map* to model these concepts.

Output of ridges or umbilics are provided via output iterator.

Approximation of ridges and umbilics are performed by two independent classes, which we now further describe.

56.4.1 Ridge Approximation

The main class is *Ridge_approximation<TriangulatedSurfaceMesh,Vertex2FTPropertyMap,Vertex2VectorPropertyMap>* . Its construction requires the mesh and the property maps defining the differential quantities for principal curvatures k_1 and k_2, the third order extremalities b_0 and b_3, the principal directions of curvature d_1 and d_2, and the fourth order quantities P_1 and P_2 if the tagging of ridges as elliptic or hyperbolic is to be done using the polynomials P_1 and P_2.

Three functions (provided as members and also as global functions) enable the computation of different types of ridges :

- *compute_max_ridges* (resp. *compute_min_ridges*) outputs ridges of types *MAX_ELLIPTIC_RIDGE* and *MAX_HYPERBOLIC_RIDGE* (resp. *MIN_ELLIPTIC_RIDGE* and *MIN_HYPERBOLIC_RIDGE*).
- *compute_crest_ridges* outputs ridges of types *MAX_CREST_RIDGE* and *MIN_CREST_RIDGE*.

These functions allows the user to specify how the elliptic/hyperbolic tagging is carried out. Notice the rationale for the choice of these three functions is simple: each computation needs a single pass over the triangles of the mesh. This should be clear for the min and max ridges. For crests, just notice max and min crests cannot intersect over a triangle.

The ridge lines are stored in *Ridge_line* objects and output through an iterator. Each ridge line is represented as a list of halfedges of the mesh it crosses with a scalar defining the barycentric coordinate of the crossing point with respect to the half-egde endpoints. Each ridge line comes with its type *Ridge_type*, its strength and sharpness.

If one chooses to use only third order quantities, the quantities P_i do not have to be defined. Then the sharpness will not be defined.

56.4.2 Umbilic Approximation

The main class is *Umbilic_approximation<TriangulatedSurfaceMesh,Vertex2FTPropertyMap,Vertex2VectorPropertyMap>* . Its construction requires the mesh and the property maps defining the differential quantities for principal curvatures k_1 and k_2, and the principal directions of curvature d_1 and d_2. The member function *compute* (or the global function *compute_umbilics*) has a parameter to define the size of the neighborhood of the umbilic.

Umbilics are stored in *Umbilic* objects, they come with their type : *ELLIPTIC_UMBILIC*, *HYPERBOLIC_UMBILIC* or *NON_GENERIC_UMBILIC*; the vertex of the mesh they are associated to and the list of half-edges representing the contour of the neighborhood.

56.4.3 Models for the Property Map Concepts

The class *Vertex2Data_Property_Map_with_std_map<TriangulatedSurfaceMesh>* enables the definition of models for the concepts *Vertex2FTPropertyMap* and *Vertex2VectorPropertyMap* using *std::maps* and *boost::associative_property_map*.

56.5 Examples

56.5.1 Example program

The following program computes ridges and umbilics from an off file.[3] It uses the Jet fitting package to estimate the differential quantities. The default output file gives rough data for visualization purpose, a verbose output file may also be asked for. Parameters are

- d, the degree of the jet for the *Monge_via_jet_fitting* class, d must be greater or equal to 3 which is the default value;
- m, the degree of the Monge representation for the *Monge_via_jet_fitting* class, m must be 3 (the default value) or 4 and smaller than d;
- a, the number of rings of neighbors collected for the *Monge_via_jet_fitting* class, in addition the number of vertices collected must be greater than $Nd := (d+1)(d+2)/2$ to make the approximation possible. a may be an integer greater than 1, the value 0 (which is the default) means that the minimum number of rings is collected to make the approximation possible. (Alternatively option p allows the specification of a constant number of neighbors);
- t, the *CGAL::Ridge_order* for the distinction between elliptic and hyperbolic ridges, t is 3 (default) or 4;
- u, the parameter for umbilic patch size, $u \geq 1$ (default is 1).

```
#include <CGAL/Ridges.h>
#include <CGAL/Umbilics.h>

//this is an enriched Polyhedron with facets' normal
#include "PolyhedralSurf.h"

typedef PolyhedralSurf::Traits          Kernel;
typedef Kernel::FT                      FT;
typedef Kernel::Point_3                 Point_3;
typedef Kernel::Vector_3                Vector_3;
```

[3]Model data may be downloaded via ftp://ftp.mpi-sb.mpg.de/pub/outgoing/CGAL/Ridges_3_datafiles.tgz . The mechanical part model has been provided courtesy of Dassault System to produce Figure 56.6, due to copyright issues the available model is not the same, it is provided by the AIM@SHAPE Shape Repository.

```
typedef PolyhedralSurf::Vertex_const_handle   Vertex_const_handle;
typedef PolyhedralSurf::Vertex_const_iterator Vertex_const_iterator;

typedef CGAL::Vertex2Data_Property_Map_with_std_map<PolyhedralSurf>
   Vertex2Data_Property_Map_with_std_map;
typedef Vertex2Data_Property_Map_with_std_map::Vertex2FT_map
   Vertex2FT_map;
typedef Vertex2Data_Property_Map_with_std_map::Vertex2Vector_map
   Vertex2Vector_map;
typedef Vertex2Data_Property_Map_with_std_map::Vertex2FT_property_map
   Vertex2FT_property_map;
typedef Vertex2Data_Property_Map_with_std_map::Vertex2Vector_property_map
    Vertex2Vector_property_map;

//RIDGES
typedef CGAL::Ridge_line<PolyhedralSurf> Ridge_line;
typedef CGAL::Ridge_approximation < PolyhedralSurf,
                                    back_insert_iterator<
   std::vector<Ridge_line*> >,
                                    Vertex2FT_property_map,
                                    Vertex2Vector_property_map >
   Ridge_approximation;
//UMBILICS
typedef CGAL::Umbilic<PolyhedralSurf> Umbilic;
typedef CGAL::Umbilic_approximation < PolyhedralSurf,
                                      back_insert_iterator<
   std::vector<Umbilic*> >,
                                      Vertex2FT_property_map,
                                      Vertex2Vector_property_map >
   Umbilic_approximation;

//create property maps
Vertex2FT_map vertex2k1_map, vertex2k2_map,
  vertex2b0_map, vertex2b3_map,
  vertex2P1_map, vertex2P2_map;
Vertex2Vector_map vertex2d1_map, vertex2d2_map;

Vertex2FT_property_map vertex2k1_pm(vertex2k1_map),
   vertex2k2_pm(vertex2k2_map),
  vertex2b0_pm(vertex2b0_map), vertex2b3_pm(vertex2b3_map),
  vertex2P1_pm(vertex2P1_map), vertex2P2_pm(vertex2P2_map);
Vertex2Vector_property_map vertex2d1_pm(vertex2d1_map),
   vertex2d2_pm(vertex2d2_map);

int main(int argc, char *argv[])
{
  //compute differential quantities with the jet fitting package
        ...
  //initialize the property maps
        ...

   //---------------------------------------------------------------------------
  //Ridges
```

```
  //---------------------------------------------------------------------------
 Ridge_approximation ridge_approximation(P,
                                         vertex2k1_pm, vertex2k2_pm,
                                         vertex2b0_pm, vertex2b3_pm,
                                         vertex2P1_pm, vertex2P2_pm,
                                         vertex2d1_pm, vertex2d2_pm);
 std::vector<Ridge_line*> ridge_lines;
 back_insert_iterator<std::vector<Ridge_line*> > ii(ridge_lines);

 //Find MAX_RIDGE, MIN_RIDGE, CREST or all ridges
 ridge_approximation.compute_max_ridges(ii, tag_order);
 ridge_approximation.compute_min_ridges(ii, tag_order);
 ridge_approximation.compute_crest_ridges(ii, tag_order);


  //----------------------------------------------------------------------------
 // UMBILICS

  //---------------------------------------------------------------------------
 Umbilic_approximation umbilic_approximation(P,
                                             vertex2k1_pm, vertex2k2_pm,
                                             vertex2d1_pm,
  vertex2d2_pm);
 std::vector<Umbilic*> umbilics;
 back_insert_iterator<std::vector<Umbilic*> > umb_it(umbilics);
 umbilic_approximation.compute(umb_it, umb_size);
}
```

56.5.2 Example: Ridges and Umbilics on an Ellipsoid

For Figure56.5, the data have been computed as follows:

```
./Compute_Ridges_Umbilics -f data/ellipsoid_u_0.02.off -d 4 -m 4 -a 3 -t
   3
```

In addition, the four elliptic umbilics are detected, the standard output being

```
nb of umbilics 4
Umbilic at location (-0.80899 0.00426003 0.293896) of type elliptic
Umbilic at location (-0.811197 0.0122098 -0.292259) of type elliptic
Umbilic at location (0.808372 -0.00551307 -0.29431) of type elliptic
Umbilic at location (0.81413 0.0018689 0.290339) of type elliptic
```

56.5.3 Example: Filtering of Crest Ridges on a Mechanical Par

Figure 56.6 illustrates the filtering of crest ridges on a mechanical model, and has been computed as follows:

```
./Compute_Ridges_Umbilics -f data/mecanic.off -d 4 -m 4 -a 4 -t 4
```

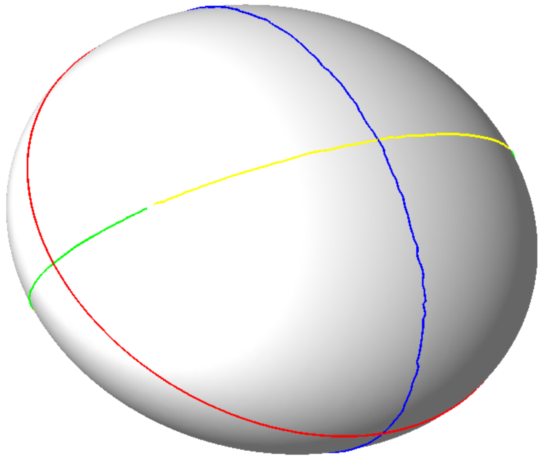


Figure 56.5: Ridges on the ellipsoid, normals pointing outward. Color coding : *MAX_ELLIPTIC_RIDGE* are blue, *MAX_HYPERBOLIC_RIDGE* are green, *MIN_ELLIPTIC_RIDGE* are red and *MIN_HYPERBOLIC_RIDGE* are yellow.

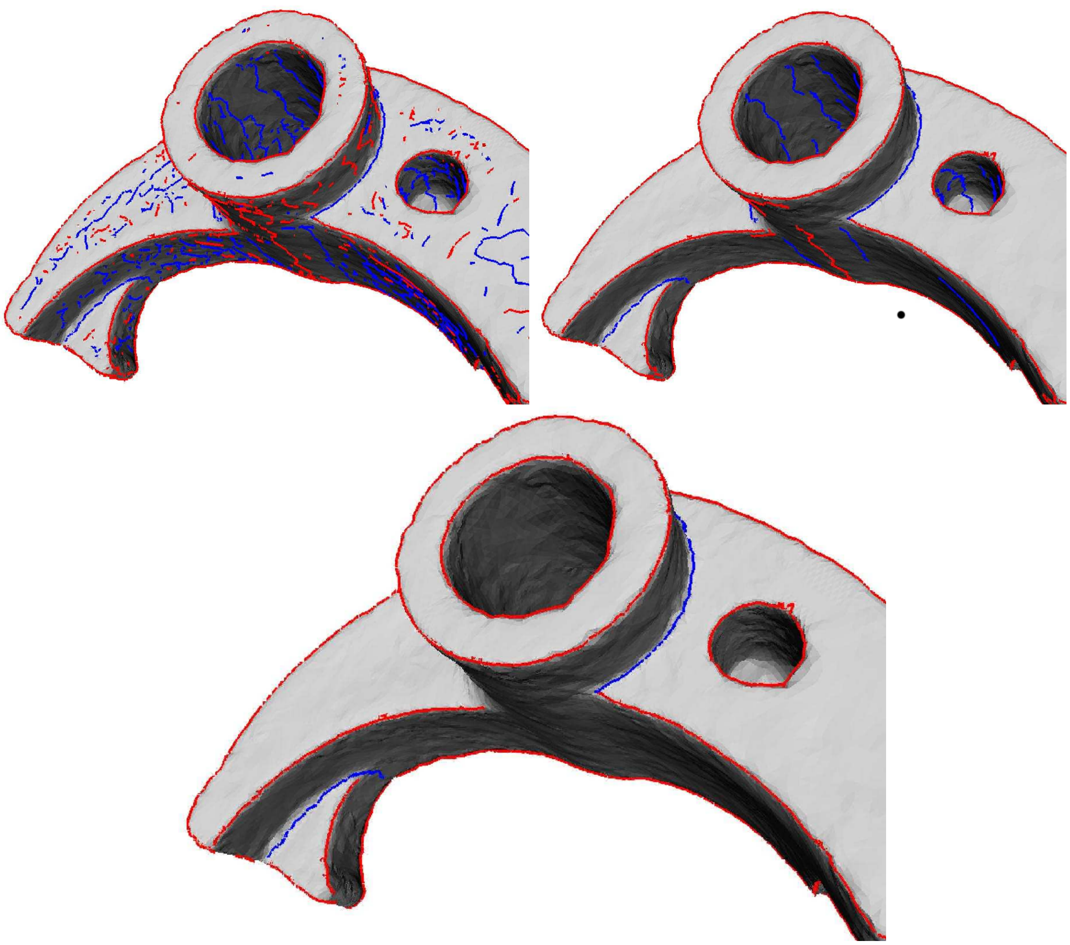


Figure 56.6: Mechanical part (37k pts): (a) All crest lines, (b) crests filtered with the strength threshold 1 and (c) crests filtered with the sharpness threshold 100 000. Notice that any point on a flat or cylindrical part lies on two ridges, so that the noise observed on the top two Figs. is unavoidable. It is however easily filtered out with the sharpness on the bottom figure.

Ridges and Umbilics Extraction on Meshes
Reference Manual

Marc Pouget and Frédéric Cazals

Concepts

Enums

Classes

Global Functions

The insert operator $<<$ is overloaded for the classes *Ridge_line* and *Umbilic*.

56.6 Alphabetical List of Reference Pages

CGAL::compute_crest_ridges

Function

Definition

The function *compute_crest_ridges* is a short cut to the method of the same name of the class *Ridge_approximation*.

#include <CGAL/Ridges.h>

template < class TriangulatedSurfaceMesh, class Vertex2FTPropertyMap, class Vertex2VectorPropertyMap, class OutputIterator>
OutputIterator compute_crest_ridges(TriangulatedSurfaceMesh P,
Vertex2FTPropertyMap vertex2k1_pm,
Vertex2FTPropertyMap vertex2k2_pm,
Vertex2FTPropertyMap vertex2b0_pm,
Vertex2FTPropertyMap vertex2b3_pm,
Vertex2VectorPropertyMap vertex2d1_pm,
Vertex2VectorPropertyMap vertex2d2_pm,
Vertex2FTPropertyMap vertex2P1_pm,
Vertex2FTPropertyMap vertex2P2_pm,
OutputIterator it,
CGAL::Ridge_order order = CGAL::Ridge_order_3)

See Also

Ridge_approximation

Function

CGAL::compute_max_ridges

Definition

The function *compute_max_ridges* is a short cut to the method of the same name of the class *Ridge_approximation*.

#include <CGAL/Ridges.h>

template < class TriangulatedSurfaceMesh, class Vertex2FTPropertyMap, class Vertex2VectorPropertyMap, class OutputIterator>
OutputIterator compute_max_ridges(TriangulatedSurfaceMesh P,
Vertex2FTPropertyMap vertex2k1_pm,
Vertex2FTPropertyMap vertex2k2_pm,
Vertex2FTPropertyMap vertex2b0_pm,
Vertex2FTPropertyMap vertex2b3_pm,
Vertex2VectorPropertyMap vertex2d1_pm,
Vertex2VectorPropertyMap vertex2d2_pm,
Vertex2FTPropertyMap vertex2P1_pm,
Vertex2FTPropertyMap vertex2P2_pm,
OutputIterator it,
CGAL::Ridge_order order = CGAL::Ridge_order_3)

See Also

Ridge_approximation

CGAL::compute_min_ridges

Definition

The function *compute_min_ridges* is a short cut to the method of the same name of the class *Ridge_approximation*.

#include <CGAL/Ridges.h>

template < class TriangulatedSurfaceMesh, class Vertex2FTPropertyMap, class Vertex2VectorPropertyMap, class OutputIterator>

```
OutputIterator    compute_min_ridges( TriangulatedSurfaceMesh P,
                                      Vertex2FTPropertyMap vertex2k1_pm,
                                      Vertex2FTPropertyMap vertex2k2_pm,
                                      Vertex2FTPropertyMap vertex2b0_pm,
                                      Vertex2FTPropertyMap vertex2b3_pm,
                                      Vertex2VectorPropertyMap vertex2d1_pm,
                                      Vertex2VectorPropertyMap vertex2d2_pm,
                                      Vertex2FTPropertyMap vertex2P1_pm,
                                      Vertex2FTPropertyMap vertex2P2_pm,
                                      OutputIterator it,
                                      CGAL::Ridge_order order = CGAL::Ridge_order_3)
```

See Also

Ridge_approximation

Function

CGAL::compute_umbilics

Definition

The function *compute_umbilics* is a short cut to the method *compute* of the class *Umbilic_approximation*.

#include <CGAL/Umbilics.h>

template < class TriangulatedSurfaceMesh, class Vertex2FTPropertyMap, class Vertex2VectorPropertyMap, class OutputIterator>

```
OutputIterator      compute_umbilics( TriangulatedSurfaceMesh P,
                                      Vertex2FTPropertyMap vertex2k1_pm,
                                      Vertex2FTPropertyMap vertex2k2_pm,
                                      Vertex2VectorPropertyMap vertex2d1_pm,
                                      Vertex2VectorPropertyMap vertex2d2_pm,
                                      OutputIterator it,
                                      double size)
```

See Also

Umbilic_approximation

CGAL::Ridge_approximation<TriangulatedSurfaceMesh,Vertex2FTPropertyMap,Ver

Class

Definition

The class *Ridge_approximation<TriangulatedSurfaceMesh,Vertex2FTPropertyMap,Vertex2VectorPropertyMap>* computes the approximation of ridges of a triangular polyhedral surface.

#include <CGAL/Ridges.h>

Parameters

The class *Ridge_approximation<TriangulatedSurfaceMesh,Vertex2FTPropertyMap,Vertex2VectorPropertyMap>* has three template parameters. *TriangulatedSurfaceMesh* provides the surface. Parameters *Vertex2FTPropertyMap* and *Vertex2VectorPropertyMap* provide the differential properties of the surface associated to its vertices.

Requirements (checked at compile time) : the types *TriangulatedSurfaceMesh::Traits::FT* and *Vertex2FTPropertyMap::value_type* must coincide; the types *TriangulatedSurfaceMesh::Traits::Vector_3* and *Vertex2VectorPropertyMap::value_type* must coincide; the types *TriangulatedSurfaceMesh::Vertex_handle*, *Vertex2FTPropertyMap::key_type* and *Vertex2VectorPropertyMap::key_type* must coincide;

Types

enum Tag_order { *Tag_3, Tag_4*};	Order of differential quantities used to distinguish elliptic and hyperbolic ridges. Third (*Tag_3*) or fourth (*Tag_4*) order quantities may be used as explained in section 56.2 of the user manual.

Creation

Ridge_approximation<TriangulatedSurfaceMesh,Vertex2FTPropertyMap,Vertex2VectorPropertyMap> ridge_approximation(TriangulatedSurfaceMesh P,

Vertex2FTPropertyMap vertex2k1_pm,

Vertex2FTPropertyMap vertex2k2_pm,

Vertex2FTPropertyMap vertex2b0_pm,

Vertex2FTPropertyMap vertex2b3_pm,

Vertex2VectorPropertyMap vertex2d1_pm,

Vertex2VectorPropertyMap vertex2d2_pm,

Ver-

tex2FTPropertyMap vertex2P1_pm,

Ver-

tex2FTPropertyMap vertex2P2_pm)

The two last property maps may not be used if computations are performed with the parameter *Tag_3*, in which case these property maps shall be initialized with their default constructors.

template <class OutputIterator>
OutputIterator ridge_approximation.compute_max_ridges(OutputIterator it, Tag_order ord = Tag_3)

Outputs ridges of types *MAX_ELLIPTIC_RIDGE* and *MAX_HYPERBOLIC_RIDGE*. Parameter *it* is a STL output iterator whose *value_type* is *Ridge_line**.

template <class OutputIterator>
OutputIterator ridge_approximation.compute_min_ridges(OutputIterator it, Tag_order ord = Tag_3)

Outputs ridges of types *MIN_ELLIPTIC_RIDGE* and *MIN_HYPERBOLIC_RIDGE*. Parameter *it* is a STL output iterator whose *value_type* is *Ridge_line**.

template <class OutputIterator>
OutputIterator ridge_approximation.compute_crest_ridges(OutputIterator it, Tag_order ord = Tag_3)

Outputs ridges of types *MAX_CREST_RIDGE* and *MIN_CREST_RIDGE*. Parameter *it* is a STL output iterator whose *value_type* is *Ridge_line**.

See Also

Ridge_line

CGAL::Ridge_line<TriangulatedSurfaceMesh>

Definition

The class *Ridge_line<TriangulatedSurfaceMesh>* stores the description of a ridge line. The list of halfedges defines a connected sequence of edges (but not as oriented halfedges). The scalar b paired with a halfedge pq is the barycentric coordinate of the crossing point r with the ridge : $r = b \times p + (1 - b) \times q$.

#include <CGAL/Ridges.h>

Types

typedef typename TriangulatedSurfaceMesh::Traits::FT

FT;

typedef typename TriangulatedSurfaceMesh::Halfedge_handle

Halfedge_handle;

typedef std::pair< Halfedge_handle, FT>

Ridge_halfhedge; A halfedge crossed by a ridge is paired with the barycentric coordinate of the crossing point.

Creation

Ridge_line<TriangulatedSurfaceMesh> ridge_line;

default constructor.

Access Functions

const Ridge_type	*ridge_line.line_type()*
const FT	*ridge_line.strength()*
const FT	*ridge_line.sharpness()*
*const std::list<Ridge_halfhedge>**	*ridge_line.line()*

The opertor $<<$ is overloaded for this class and returns the line type, strength, sharpness and coordinates of the points of the polyline.

See Also

Ridge_approximation,

CGAL::Ridge_order

Definition

The enum *Ridge_order* describes the order of differential quantities used to distinguish elliptic and hyperbolic ridges. Third or fourth order quantities may be used as explained in section 56.2 of the user manual.

#include <CGAL/Ridges.h>

enum Ridge_order { Ridge_order_3 = 3, Ridge_order_4};

See Also

Ridge_approximation

CGAL::Ridge_type

Definition

The enum *Ridge_type* describes the types for the class *Ridge_line*.

#include <CGAL/Ridges.h>

enum Ridge_type { *MAX_ELLIPTIC_RIDGE = 1,*
MAX_HYPERBOLIC_RIDGE,
MAX_CREST_RIDGE,
MIN_ELLIPTIC_RIDGE,
MIN_HYPERBOLIC_RIDGE,
MIN_CREST_RIDGE}

See Also

Ridge_line

Concept

TriangulatedSurfaceMesh

Definition

The concept TriangulatedSurfaceMesh describes an oriented surface (possibly with boundary) embedded in the 3 space. It is combinatorially based on a halfedge data structure with triangular faces, geometrically 3d points associated to the vertices define the embedding.

Types

TriangulatedSurfaceMesh:: Traits	geometric Traits, this is a CGAL::Kernel concept.
TriangulatedSurfaceMesh:: Facet	Opaque type representing a facet of the TriangulatedSurfaceMesh .
TriangulatedSurfaceMesh:: Facet_handle	Handle to a facet. Model of the Handle concept.
TriangulatedSurfaceMesh:: Facet_const_handle	
TriangulatedSurfaceMesh:: Facet_iterator	Iterator over all mesh facets. Model of the ForwardIterator concept.
TriangulatedSurfaceMesh:: Facet_const_iterator	
TriangulatedSurfaceMesh:: Vertex	Opaque type representing a vertex of the TriangulatedSurfaceMesh .
TriangulatedSurfaceMesh:: Vertex_handle	Handle to a vertex. Model of the Handle concept.
TriangulatedSurfaceMesh:: Vertex_const_handle	
TriangulatedSurfaceMesh:: Vertex_iterator	Iterator over all vertices of a mesh. Model of the ForwardIterator concept.
TriangulatedSurfaceMesh:: Vertex_const_iterator	
TriangulatedSurfaceMesh:: Halfedge	Opaque type representing a halfedge of the TriangulatedSurfaceMesh .
TriangulatedSurfaceMesh:: Halfedge_handle	Handle to a halfedge. Model of the Handle concept.
TriangulatedSurfaceMesh:: Halfedge_const_handle	
TriangulatedSurfaceMesh:: Halfedge_iterator	Iterator over all halfedges of a TriangulatedSurfaceMesh . Model of the ForwardIterator concept.
TriangulatedSurfaceMesh:: Halfedge_const_iterator	
TriangulatedSurfaceMesh:: Point_iterator	Iterator over all points of a TriangulatedSurfaceMesh , its value type is *Traits::Point_3*. Model of the ForwardIterator concept.
TriangulatedSurfaceMesh:: Point_const_iterator	

Creation

Construction and destruction are undefined.

Operations

Facet_const_iterator	*mesh.facets_begin()*	iterator over all facets (excluding holes).
Facet_const_iterator	*mesh.facets_end()*	past-the-end iterator.
Point_iterator	*mesh.points_begin()*	iterator over all points.

Point_iterator	*mesh.points_end()*	past-the-end iterator.

Methods for the Vertex :

Traits::Point_3&	*vertex.point()*	The point associated to the vertex

Methods for the Halfedge :

Halfedge_const_handle	*halfedge.opposite() const*	the opposite halfedge.
Halfedge_const_handle	*halfedge.next() const*	the next halfedge around the facet.
Halfedge_const_handle	*halfedge.prev() const*	the previous halfedge around the facet.
bool	*halfedge.is_border_edge() const*	is true if *halfedge* or *halfedge.opposite()* is a border halfedge.
Vertex_const_handle	*halfedge.vertex() const*	the incident vertex of *halfedge*.
Facet_const_handle	*halfedge.facet() const*	the incident facet of *halfedge*. If *halfedge* is a border halfedge the result is default construction of the handle.

Methods for the Facet :

Halfedge_const_handle	*facet.halfedge() const*	an incident halfedge that points to *facet*.

Has Models

CGAL::Polyhedron_3 with the restriction that faces are triangular.

Class

CGAL::Umbilic<TriangulatedSurfaceMesh>

Definition

The class *Umbilic<TriangulatedSurfaceMesh>* stores the description of an umbilic.

#include <CGAL/Umbilics.h>

Types

typedef typename TriangulatedSurfaceMesh::Vertex_handle

Vertex_handle;

typedef typename TriangulatedSurfaceMesh::Halfedge_handle

Halfedge_handle;

Creation

Access Functions

const Vertex_handle *umbilic.vertex()*

const Umbilic_type *umbilic.umbilic_type()*

std::list<Halfedge_handle>

umbilic.contour_list()

Operations

The usual insert operator (<<) is overloaded for *Umbilic*, it gives the location (3d coordinates of the vertex) and the type.

See Also

Umbilic_approximation

CGAL::Umbilic_approximation<TriangulatedSurfaceMesh,Vertex2FTPropertyMap,V

Class

Definition

The class *Umbilic_approximation<TriangulatedSurfaceMesh,Vertex2FTPropertyMap,Vertex2VectorPropertyMap>* computes the approximation of umbilics on a triangular polyhedral surface.

#include <CGAL/Umbilics.h>

Parameters

The class *Umbilic_approximation<TriangulatedSurfaceMesh,Vertex2FTPropertyMap,Vertex2VectorPropertyMap>* has three template parameters. *TriangulatedSurfaceMesh* provides the surface. Parameters *Vertex2FTPropertyMap* and *Vertex2VectorPropertyMap* provide the differential properties of the surface associated to its vertices.

Requirements (checked at compile time) : the types *TriangulatedSurfaceMesh::Traits::FT* and *Vertex2FTPropertyMap::value_type* must coincide; the types *TriangulatedSurfaceMesh::Traits::Vector_3* and *Vertex2VectorPropertyMap::value_type* must coincide; the types *TriangulatedSurfaceMesh::Vertex_handle*, *Vertex2FTPropertyMap::key_type* and *Vertex2VectorPropertyMap::key_type* must coincide;

Types

typedef typename TriangulatedSurfaceMesh::Traits::FT

FT;

Creation

Umbilic_approximation<TriangulatedSurfaceMesh,Vertex2FTPropertyMap,Vertex2VectorPropertyMap>
umbilic_approximation(TriangulatedSurfaceMesh P, *Ve*

tex2FTPropertyMap vertex2k1_pm, *Ve*

tex2FTPropertyMap vertex2k2_pm, *Ve*

tex2VectorPropertyMap vertex2d1_pm, *Ve*

tex2VectorPropertyMap vertex2d2_pm)

default constructor.

Operations

template <class OutputIterator>

OutputIterator *umbilic_approximation.compute(OutputIterator it, FT size)*

Performs the approximation, *size* determines the size of the patches around vertices, taken as *size* times the size of the 1-ring. Umbilics are output through the *OutputIterator* which is a concept of output iterator of the STL with value type *Umbilic**.

See Also

Umbilic TriangulatedSurfaceMesh Vertex2FTPropertyMap Vertex2VectorPropertyMap

CGAL::Umbilic_type

Definition

The enum *Umbilic_type* describes the types for the class *Umbilic*.

#include <CGAL/Umbilics.h>

enum Umbilic_type { *NON_GENERIC_UMBILIC, ELLIPTIC_UMBILIC, HYPERBOLIC_UMBILIC*}*;*

See Also

Umbilic.

Class

CGAL::Vertex2Data_Property_Map_with_std_map< TriangulatedSurfaceMesh>

Definition

The class *Vertex2Data_Property_Map_with_std_map<TriangulatedSurfaceMesh>* provides models for the concepts *Vertex2FTPropertyMap* and *Vertex2VectorPropertyMap* to be used for *Ridge_approximation*. The property maps are created with the *boost::associative_property_map* adaptor from *std::map*.

#include <CGAL/Ridges.h>

Types

typedef typename TriangulatedSurfaceMesh::Traits::FT

FT;

typedef typename TriangulatedSurfaceMesh::Traits::Vector_3

Vector_3;

typedef typename TriangulatedSurfaceMesh::Vertex_handle

Vertex_handle;

struct Vertex_cmp { bool operator();};

typedef std::map<Vertex_handle, FT, Vertex_cmp>

Vertex2FT_map;

typedef boost::associative_property_map< Vertex2FT_map >

Vertex2FT_property_map;

typedef std::map<Vertex_handle, Vector_3, Vertex_cmp>

Vertex2Vector_map;

typedef boost::associative_property_map< Vertex2Vector_map >

Vertex2Vector_property_map;

Is Model for the Concepts

Vertex2FT_property_map is model of *Vertex2FTPropertyMap*.

Vertex2Vector_property_map is model of *Vertex2VectorPropertyMap*.

See Also

Ridge_approximation

Vertex2FTPropertyMap

Concept

Definition

The concept Vertex2FTPropertyMap specializes the concept of LvaluePropertyMap of the Boost library. It is intended to be used in combination with the concept *TriangulatedSurfaceMesh* in the class *Ridge_approximation*. It associates a field type value *TriangulatedSurfaceMesh::Traits::FT* to keys which are *TriangulatedSurfaceMesh::Vertex_handle*.

Has Models

Vertex2Data_Property_Map_with_std_map::Vertex2FT_property_map,

See Also

TriangulatedSurfaceMesh.

Concept

Vertex2VectorPropertyMap

Definition

The concept Vertex2VectorPropertyMap specializes the concept of LvaluePropertyMap of the Boost library. It is intended to be used in combination with the concept *TriangulatedSurfaceMesh* in the class *Ridge_approximation*. It associates a three dimensional vector *TriangulatedSurfaceMesh::Traits::Vector_3* to keys which are *TriangulatedSurfaceMesh::Vertex_handle*.

Has Models

Vertex2Data_Property_Map_with_std_map::Vertex2Vector_property_map,

See Also

TriangulatedSurfaceMesh.

Chapter 57

Estimation of Local Differential Properties of Point-Sampled Surfaces

Marc Pouget and Frédéric Cazals

Contents

This package allows the estimation of local differential quantities of a surface from a point sample, given either as a mesh or as point cloud.

Note that this package needs the third party library Eigen (Page 28) to be installed to compile the example code.

Figure 57.1: Principal directions associated with k_{max} scaled by k_{min}.

57.1 Introduction

57.1.1 Overview

Consider a sampled smooth surface, and assume we are given a collection of points P about a given sample p. We aim at estimating the differential properties up to any fixed order of the surface at point p from the point set $P^+ = P \cup \{p\}$ — we denote $N = | P^+ |$. More precisely, first order properties correspond to the normal or the tangent plane; second order properties provide the principal curvatures and directions, third order properties provide the directional derivatives of the principal curvatures along the curvature lines, etc. Most of the time, estimating first and second order differential quantities is sufficient. However, some applications involving shape analysis require estimating third and fourth order differential quantities. Many different estimators have been proposed in the vast literature of applied geometry [Pet01] (section 3, page 7), and all of them need to define a neighborhood around the point at which the estimation is computed. Our method relies on smooth differential geometry calculations, carried out on smooth objects *fitted* from the sample points. Datasets amenable to such a processing are naturally unstructured point clouds, as well as meshes —whose topological information may be discarded.

Estimating differential properties from discrete date always raises a philosophical issue. On one hand, estimating differential quantities subsumes a smooth surface does exist. In this spirit one wishes to recover its differential properties, so that any estimation method must come with an asymptotic convergence analysis of the results returned. For the method developed in this CGAL package, the interested will find such an analysis in [CP05a], (Theorem 3)— it should be stressed the error bounds proved therein are optimal.

On the other hand, any estimation method may be applied to arbitrarily data —surface unknown, surface piecewise smooth etc. In such a case, no analysis can be carried out, and it is up to the users to check the results match their needs.

Unlike most of the CGAL packages, this package uses approximation methods and is not intended to provide an exact canonical result in any sense. This is why internal computations are performed with a number type possibly different from that of the input data, even if for convenience the results are returned with this original number type. A reasonable choice for this internal number type is for example the *double* type.

57.1.2 Smooth Surfaces, d-Jets and the Monge Form

To present the method, we shall need the following notions. Consider a smooth surface. About one of its points, consider a coordinate system whose z-axis does not belong to the tangent space. In such a frame, the surface can locally be written as the graph of a bivariate function. Letting $h.o.t.$ stand for *higher order terms*, one has :

$$z(x,y) = J_{B,d}(x,y) + h.o.t. \ ; \quad J_{B,d}(x,y) = \sum_{k=0}^{d} \Big(\sum_{i=0}^{i} \frac{B_{k-i,i}x^{k-i}y^{i}}{i!(k-i)!}\Big). \tag{57.1}$$

The degree d polynomial $J_{B,d}$ is the Taylor expansion of the function z, and is called its *d-jet*. Notice that a d-jet contains $N_d = (d+1)(d+2)/2$ coefficients.

Recall that an umbilical point of a surface — or umbilic for short, is a point where both principal curvatures are identical. At any point of the surface which is not an umbilic, principal directions d_1, d_2 are well defined, and these (non oriented) directions together with the normal vector n define two direct orthonormal frames. If v_1 is a unit vector of direction d_1, there exists a unique unit vector v_2 so that (v_1, v_2, n) is direct; and the other possible frame is $(-v_1, -v_2, n)$. Both these coordinate systems are known as the *Monge coordinate systems*. In both these systems, the surface is said to be given in the Monge form and its jet has the following canonical form :

$$z(x,y) = \quad \tfrac{1}{2}(k_1x^2 + k_2y^2) + \tfrac{1}{6}(b_0x^3 + 3b_1x^2y + 3b_2xy^2 + b_3y^3) \tag{57.2}$$

$$+ \tfrac{1}{24}(c_0x^4 + 4c_1x^3y + 6c_2x^2y^2 + 4c_3xy^3 + c_4y^4) + h.o.t. \tag{57.3}$$

The coefficients k_1, k_2 are the principal curvatures, b_0, b_3 are the directional derivatives of k_1, k_2 along their respective curvature line, while b_1, b_2 are the directional derivatives of k_1, k_2 along the other curvature lines.

The Monge coordinate system can be computed from any d-jet ($d \geq 2$), and so are the Monge coefficients. These informations characterize the local geometry of the surface in a canonical way, and are the quantities returned by our algorithm.

57.1.3 Algorithm

Based on the above concepts, the algorithm consists of 4 steps.

1. We perform a Principal Component Analysis (PCA) on P^+. This analysis outputs three orthonormal eigenvectors and the associated eigenvalues. The fitting basis consists of these three vectors so that the vector associated to the smallest eigenvalue is the last vector of the basis. (Indeed, if the surface is well sampled, one expects the PCA to provide one small and two large eigenvalues, the eigenvector associated to the small one approximating the normal vector.)

2. We perform a change of coordinates to move the samples into the coordinate system of the fitting basis and with origin the point p at which the estimation is sought. We then resort to polynomial fitting, so as to either interpolate or approximate the d-jet of the surface in this coordinate system. This bivariate polynomial approximation reduces to linear algebra operations.

3. From the fitted d-jet, we compute the Monge basis (d_1, d_2, n).

4. Finally, we compute the Monge coefficients : k_i, b_i, c_i.

Further details can be found in section 57.4 and in [CP05a] (section 6).

57.1.4 Degenerate Cases

As usual, the fitting procedure may run into (almost) degenerate cases:

- Due to poor sampling, the PCA used to determine a rough normal vector may not be good. The nearer this direction to the tangent plane the worse the estimation.
- As observed in [CP05a] (section 3.1), the interpolating problem is not *poised* if the points project, into the fitting frame, onto an algebraic curve of degree d. More generally, the problem is ill poised if the condition number is too large.

In these cases, even if a result is provided, the estimation may not be relevant. To inform the user of these issues, we provide the PCA results and the condition number of the fitting. In any case, it is up to the user to judge if the result meets its need.

57.2 Software Design

57.2.1 Options and Interface Specifications

The fitting strategy performed by the class *Monge_via_jet_fitting* requires the following parameters:

- the degree d of the fitted polynomial ($d \geq 1$),
- the degree d' of the Monge coefficients sought, with $1 \leq d' \leq \min(d, 4)$,
- a range of N input points on the surface, with the precondition that $N \geq N_d = (d+1)(d+2)/2$. Note that if $N = N_d$, interpolation is performed; and if $N > N_d$, approximation is used.

57.2.2 Output

As explained in Section 57.1, the output consists of a coordinate system, the Monge basis, together with the Monge coefficients which are stored in the *Monge_form* class. In addition, more information on the computational issues are stored in the *Monge_via_jet_fitting* class.

The *Monge_form* class provides the following information.

- Origin. This is the point on the fitted polynomial surface where the differential quantities have been computed. In the approximation case, it differs from the input point p : it is the projection of p onto the fitted surface following the z-direction of the fitting basis.

- Monge Basis. The Monge basis (d_1, d_2, n) is orthonormal direct, and the maximal, minimal curvatures are defined wrt this basis. If the user has a predefined normal n_0 (e.g. the sample points come from an oriented mesh) then if $n_0.n > 0$ then max-min is correct; if not, i.e. $n_0.n < 0$, the user should switch to the orthonormal direct basis $(d'_1, d'_2, n') = (d_2, d_1, -n)$ with the maximal curvature $k'_1 = -k_2$ and the minimal curvature $k'_2 = -k_1$. If $n_0.n = 0$ or is small, the orientation of the surface is clearly ill-defined, and the user may proof-check the samples used to comply with its predefined normal.

- Monge Coefficients. The coefficient of the Monge form is $(k_1, k_2(\leq k_1), b_0, b_1, b_2, b_3, c_0, c_1, c_2, c_3, c_4)$ for $d' = 4$.

In addition, the class *Monge_via_jet_fitting* stores

- the condition number of the fitting system,
- the coordinate system of the PCA (in which the fitting is performed).

57.2.3 Template Parameters

Template parameter `DataKernel`

This concept provides the types for the input sample points, together with $3d$ vectors and a number type. It is used as template for the class *Monge_via_jet_fitting<DataKernel, LocalKernel, SvdTraits>* . Typically, one can use *CGAL::Cartesian<double>*.

Template parameter `LocalKernel`

This is a parameter of the class *Monge_via_jet_fitting<DataKernel, LocalKernel, SvdTraits>*. This concept defines the vector and number types used for local computations and to store the PCA basis data.

Input points of type *DataKernel::Point_3* are converted to *LocalKernel::Point_3*. For output of the *Monge_form* class, these types are converted back to *Data_Kernel* ones. Typically, one can use *CGAL::Cartesian<double>* which is the default.

Template parameter `SvdTraits`

This concept provides the number, vector and matrix types for algebra operations required by the fitting method in *Monge_via_jet_fitting<DataKernel, LocalKernel, SvdTraits>* . The main method is a linear solver using a singular value decomposition.

Compatibility requirements

To solve the fitting problem, the sample points are first converted from the *DataKernel* to the *LocalKernel* (this is done using the *CGAL::Cartesian_converter*). Then change of coordinate systems and linear algebra operations are performed with this kernel. This implies that the number types *LocalKernel::FT* and *SvdTraits::FT* must be identical. Second the Monge basis and coefficients, computed with the *LocalKernel*, are converted back to the *DataKernel* (this is done using the *CGAL::Cartesian_converter* and the *CGAL::NT_converter*).

57.3 Examples

57.3.1 Single Estimation about a Point of a Point Cloud

The first example illustrates the computation of the local differential quantities from a set of points given in a text file as input. The first point of the list is the one at which the computation is performed. The user has to specify a file for the input points and the degrees d and d'.

```
#include <CGAL/Cartesian.h>


#include <fstream>
#include <vector>

#include <CGAL/Monge_via_jet_fitting.h>
typedef double                     DFT;
typedef CGAL::Cartesian<DFT>       Data_Kernel;
typedef Data_Kernel::Point_3       DPoint;
typedef CGAL::Monge_via_jet_fitting<Data_Kernel>
   My_Monge_via_jet_fitting;
typedef My_Monge_via_jet_fitting::Monge_form     My_Monge_form;

int main(int argc, char *argv[])
{
  size_t d_fitting = 4;
  size_t d_monge = 4;
  const char* name_file_in = "data/in_points_d4.txt";
  //check command line
  if (argc<4)
    {
      std::cout << " Usage : " << argv[0]
                << " <inputPoints.txt> <d_fitting> <d_monge>" <<
   std::endl
                << "test with default arguments" << std::endl;
    }
  else {
    name_file_in = argv[1];
    d_fitting = std::atoi(argv[2]);
    d_monge = std::atoi(argv[3]);
  }

  //open the input file
  std::ifstream inFile(name_file_in);
  if ( !inFile )
    {
      std::cerr << "cannot open file for input\n";
      exit(-1);
    }

  //initalize the in_points container
  double x, y, z;
  std::vector<DPoint> in_points;
  while (inFile >> x) {
```

```
    inFile >> y >> z;
    DPoint p(x,y,z);
    in_points.push_back(p);
  }
  inFile.close();
  // fct parameters

  My_Monge_form monge_form;
  My_Monge_via_jet_fitting monge_fit;
  monge_form = monge_fit(in_points.begin(), in_points.end(), d_fitting,
    d_monge);

  //OUTPUT on std::cout
  CGAL::set_pretty_mode(std::cout);
  std::cout << "vertex : " << in_points[0] << std::endl
            << "number of points used : " << in_points.size() <<
    std::endl
            << monge_form;
  std::cout  << "condition_number : " << monge_fit.condition_number() <<
    std::endl
             << "pca_eigen_vals and associated pca_eigen_vecs :"  <<
    std::endl;
  for (int i=0; i<3; i++)
    std::cout << monge_fit.pca_basis(i).first << std::endl
              << monge_fit.pca_basis(i).second  << std::endl;
  return 0;
}
```

File: examples/Jet_fitting_3/Single_estimation.cpp

57.3.2 On a Mesh

The second example (cf Mesh_estimation.cpp in the example directory) illustrates the computation of local differential quantities for all vertices of a given mesh. The neighborhood of a given vertex is computed using rings on the triangulation. Results are twofold:

- a human readable text file featuring the *Monge_form* and numerical informations on the computation : condition number and the PCA basis;
- another text file that records raw data (better for a vizualization post-processing).

Figs. 57.1 and 57.2 provide illustrations of principal directions of curvature.

advanced

57.4 Mathematical and Algorithmic Details

In this Section, we detail the mathematics involved, in order to justify the design choices made. To begin with, observe the fitting problem involves three relevant direct orthonormal basis: the world-basis (w_x, w_y, w_z), the fitting-basis (f_x, f_y, f_z), the Monge basis (d_1, d_2, n).

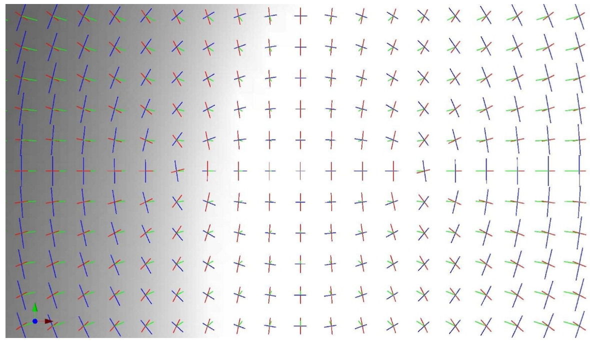


Figure 57.2: Principal directions of curvature and normals at vertices of a mesh of the graph of the function $f(x,y) = 2x^2 + y^2$.

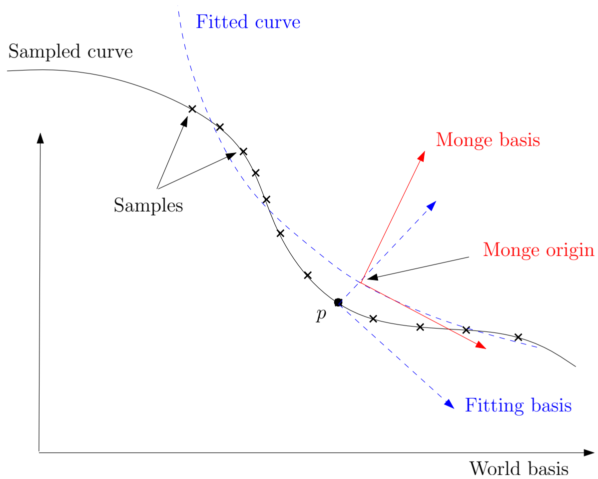


Figure 57.3: The three bases involved in the estimation.

57.4.1 Computing a Basis for the Fitting

Input : samples
Output : fitting-basis

Performing a PCA requires diagonalizing a symmetric matrix. This analysis gives an orthonormal basis whose z-axis is provided by the eigenvector associated to the smallest eigenvalue.[1] Note one may have to swap the orientation of a vector to get a direct basis.

Let us denote $P_{W \to F}$ the matrix that changes coordinates from the world-basis (w_x, w_y, w_z) to the fitting-basis (f_x, f_y, f_z). The rows of $P_{W \to F}$ are the coordinates of the vectors (f_x, f_y, f_z) in the world-basis. This matrix represents a orthogonal transformation hence its inverse is its transpose. To obtain the coordinates of a point in the fitting-basis from the coordinates in the world-basis, one has to multiply by $P_{W \to F}$.

As mentioned above, the eigenvalues are returned, from which the sampling quality can be assessed. For a good sampling, the eigenvector associated to the smallest eigenvalue should roughly give the normal direction.

[1] Another possibility is to choose as z-axis the axis of the world-basis with the least angle with the axis determined with the PCA. Then the change of basis reduces to a permutation of axis.

57.4.2 Solving the Interpolation / Approximation Problem

Input : samples, fitting-basis
Output : coefficients $A_{i,j}$ of the bivariate fitted polynomial in the fitting-basis

Computations are done in the fitting-basis and the origin is the point p. First, one has to transform coordinates of sample points with a translation $(-p)$ and multiplication by $P_{W\to F}$.

The fitting process consists of finding the coefficients $A_{i,j}$ of the degree d polynomial

$$J_{A,d} = \sum_{k=0}^{d} (\sum_{i=0}^{k} \frac{A_{k-i,i}x^{k-i}y^i}{i!(k-i)!}). \tag{57.4}$$

Denote $p_i = (x_i, y_i, z_i),\ i = 1,\ldots,N$ the coordinates of the sample points of P^+. For interpolation the linear equations to solve are $A(x_i, y_i) = z_i\ i = 1,\ldots,N$, and for approximation one has to minimize $\sum_{i=1}^{N}(A(x_i, y_i) - z_i)^2$. The linear algebra formulation of the problem is given by

$$A = \quad (A_{0,0}, A_{1,0}, A_{0,1}, \ldots, A_{0,d})^T \tag{57.5}$$

$$Z = \quad (z_1, z_2, \ldots, z_N)^T \tag{57.6}$$

$$M = \quad (1, x_i,\ y_i,\ \tfrac{x_i^2}{2}, \ldots,\ \tfrac{x_i y_i^{d-1}}{(d-1)!},\ \tfrac{y_i^d}{d!})_{i=1,\ldots,N} \tag{57.7}$$

$$\tag{57.8}$$

The equations for interpolation become $MA = Z$. For approximation, the system $MA = Z$ is solved in the least square sense, i.e. one seeks the vector A such that $A = \arg\min_A ||MA - Z||_2$.

In any case, there is a preconditioning of the matrix M so as to improve the condition number. Assuming the $\{x_i\}$, $\{y_i\}$ are of order h, the pre-conditioning consists of performing a column scaling by dividing each monomial $x_i^k y_i^l$ by h^{k+l} — refer to Eq. (57.5). Practically, the parameter h is chosen as the mean value of the $\{x_i\}$ and $\{y_i\}$. In other words, the new system is $M'Y = (MD^{-1})(DA) = Z$ with D the diagonal matrix $D = (1, h, h, h^2, \ldots, h^d, h^d)$, so that the solution A of the original system is $A = D^{-1}Y$.

There is always a single solution since for under constrained systems we also minimize $||A||_2$. The method uses a singular value decomposition of the $N \times N_d$ matrix $M = USV^T$, where U is a $N \times N$ orthogonal matrix, V is a $N_d \times N_d$ orthogonal matrix and S is a $N \times N_d$ matrix with the singular values on its diagonal. Denote r the rank of M, we can decompose $S = \begin{pmatrix} D_r & 0_{r,\ N_d-r} \\ 0_{N-r,\ r} & 0_{N-r,\ N_d-r} \end{pmatrix}$. The number r, which is the number of non zero singular values, is strictly lower than N_d if the system is under constrained. In any case, the unique solution which minimize $||A||_2$ is given by :

$$A = V \begin{pmatrix} D_r^{-1} & 0_{N_d-r,\ r} \\ 0_{r,\ N-r} & 0_{N_d-r,\ N-r} \end{pmatrix} U^T Z. \tag{57.9}$$

One can provide the condition number of the matrix M (after preconditioning) which is the ratio of the maximal and the minimal singular values. It is infinite if the system is under constrained, that is the smallest singular value is zero.

Implementation details. We assume a *solve* function is provided by the traits *SvdTraits*. This function solves the system MX=B (in the least square sense if M is not square) using a Singular Value Decomposition and gives the condition number of M.

Remark: as an alternative, other methods may be used to solve the system. A *QR* decomposition can be substituted to the *SVD*. One can also use the normal equation $M^T MX = MTB$ and apply methods for square systems such as LU, QR or Cholesky since $M^T M$ is symmetric definite positive when M has full rank. The advantages of the SVD is that it works directly on the rectangular system and gives the condition number of the system. For more on these alternatives, see [GvL83] (Chap. 5).

57.4.3 Principal Curvature / Directions

Input : coefficients of the fit $A_{i,j}$, fitting-basis
Output : Monge basis wrt fitting-basis and world-basis

In the fitting basis, we have determined a height function expressed by Eq. (57.4). Computations are done in the fitting-basis. The partial derivatives, evaluated at $(x,y) = (0,0)$, of the fitted polynomial $J_{A,d}(x,y)$ are $A_{i,j} = \frac{\partial^{i+j} J_{A,d}}{\partial^i x \partial^j y}$ Expanding Eq. (57.4) yields:

$$J_{A,d}(x,y) = A_{0,0} + A_{1,0}x + A_{0,1}y + \frac{1}{2}(A_{2,0}x^2 + 2A_{1,1}xy + A_{0,2}y^2) + \frac{1}{6}(A_{3,0}x^3 + 3A_{2,1}x^2y + \dots) + \dots \quad (57.10)$$

- The origin, that is the point of the fitted surface where the estimation is performed, is $(0,0,A_{0,0})$.
- The normal is $n = (-A_{1,0}, -A_{0,1}, 1)/\sqrt{A_{1,0}^2 + A_{0,1}^2 + 1}$.
- Curvature related properties are retrieved resorting to standard differential calculus [dC76] (Chap. 3). More precisely, the Weingarten operator $W = -I^{-1}II$ is first computed in the basis of the tangent plane $\{(1,0,A_{1,0}),(0,1,A_{0,1})\}$. We compute an orthonormal basis of the tangent plane using the Gram-Schmidt algorithm, and then we compute Weingarten in this basis (applying a change of basis with the matrix $W' = P^{-1}WP$). In this orthonormal basis, the matrix of the Weingarten map is symmetric and we diagonalize it. One finally gets the principal curvatures which are the eigenvalues of W, and the associated principal directions. This gives an orthonormal direct basis (d_1,d_2,n). Let us denote $P_{F\to M}$ the matrix to change coordinates from the fitting-basis to the Monge basis. Its rows are the coordinates of the vectors (d_1,d_2,n) in the fitting-basis. It is an orthogonal matrix $P_{F\to M}^{-1} = P_{F\to M}^T$. The Monge basis expressed in the world-basis is obtained by multiplying the coordinates of (d_1,d_2,n) in the fitting-basis by $P_{W\to F}^{-1}$, (the same holds for the origin point which has in addition to be translated by p, i.e. the coordinates of the origin point are $P_{W\to F}^{-1}(0,0,A_{0,0}) + p$.

57.4.4 Computing Higher Order Monge Coefficients

Input : coefficients of the fit, Monge basis wrt fitting-basis ($P_{F\to M}$)
Output : third and fourth order coefficients of Monge

We use explicit formula. The implicit equation of the fitted polynomial surface in the fitting-basis with origin the point $(0,0,A_{0,0})$ is $Q = 0$ with

$$Q = -w - A_{0,0} + \sum_{i,j} \frac{A_{i,j} u^i v^j}{i!j!}. \quad (57.11)$$

The equation in the Monge basis is obtained by substituting (u,v,w) by $P_{F\to M}^T(x,y,z)$. Denote $f(x,y,z) = 0$ this implicit equation. By definition of the Monge basis, we have locally (at $(0,0,0)$)

$$f(x,y,z) = 0 \Leftrightarrow z = g(x,y) \quad (57.12)$$

and the Taylor expansion of g at $(0,0)$ are the Monge coefficients sought. Let us denote the partial derivatives evaluated at the origin of f and g by $f_{i,j,k} = \frac{\partial^{i+j+k} f}{\partial^i x \partial^j y \partial^k z}$ and $g_{i,j} = \frac{\partial^{i+j} g}{\partial^i x \partial^j y}$. One has $f_{1,0,0} = f_{0,1,0} = f_{1,1,0} = 0$, $g_{0,0} = g_{1,0} = g_{0,1} = g_{1,1} = 0$ and $g_{2,0} = k_1$, $g_{0,2} = k_2$. The partial derivative of order n of f depends on the matrix $P_{F\to M}$ and the partial derivatives of order at most n of $J_{A,d}$. The third and fourth order coefficients of are computed with the implicit function theorem. For instance :

$$
\begin{array}{c}
b_0 = g_{3,0} = -\frac{f_{3,0,0}f_{0,0,1}-3f_{1,0,1}f_{2,0,0}}{f_{0,0,1}^2} \\
b_1 = g_{2,1} = -\frac{-f_{0,1,1}f_{2,0,0}+f_{2,1,0}f_{0,0,1}}{f_{0,0,1}^2} \\
\ldots.
\end{array}
$$

advanced

Estimation of Local Differential Properties of Sampled Surfaces via Polynomial Fitting
Reference Manual

Marc Pouget and Frédéric Cazals

Requirements

Concepts

Classes

Global Functions

The insert operator (*operator<<*) is overloaded for the class *Monge_form*.

57.5 Alphabetical List of Reference Pages

DataKernel

Definition

The concept DataKernel describes the set of requirements to be fulfilled by any class used to instantiate first template parameter of the class *Monge_via_jet_fitting<DataKernel,LocalKernel,SvdTraits>*.

Types

DataKernel:: FT	The scalar type.
DataKernel:: Point_3	The point type.
DataKernel:: Vector_3	The vector type.

Operations

Only constructors (from 3 scalars and copy constructors) and access methods to coordinates *x()*, *y()*, *z()* are needed.

Has Models

CGAL::Cartesian<FieldNumberType>

See Also

The *LocalKernel* concept.

Class

CGAL::Eigen_svd

Definition

The class *Eigen_svd* provides an algorithm to solve in the least square sense a linear system with a singular value decomposition using Eigen (Page 28). The field type is *double*.

#include <CGAL/Eigen_svd.h>

Is Model for the Concepts

SvdTraits

LocalKernel

Concept

Definition

The concept LocalKernel describes the set of requirements to be fulfilled by any class used to instantiate the second template parameter of the class *Monge_via_jet_fitting<DataKernel,LocalKernel,SvdTraits>*.

This concept provides the geometric primitives used for the computations in the class *Monge_via_jet_fitting*.

Requirements

In the class *Monge_via_jet_fitting* the scalar type, *LocalKernel::FT*, must be the same as that of the *SvdTraits* concept : *SvdTraits::FT*.

The type *LocalKernel::FT* is a model of the FieldWithSqrt concept.

Types

LocalKernel:: FT	The scalar type.
LocalKernel:: Point_3	The point type.
LocalKernel:: Vector_3	The vector type.

Operations

The scalar type *LocalKernel::FT* must be a field type with a square root.

Only constructors (from 3 scalars and copy constructors) and access methods to coordinates *x()*, *y()*, *z()* are needed for the point and vector types.

Has Models

CGAL::Cartesian<FieldNumberType>

See Also

The *DataKernel* and *SvdTraits* concepts.

CGAL::Monge_via_jet_fitting< DataKernel, LocalKernel, SvdTraits> ::Monge_form

Definition

The class *Monge_form* stores the Monge representation, i.e., the Monge coordinate system and the coefficients of the Monge form in this system.

#include <CGAL/Monge_via_jet_fitting.h>

Types

typedef typename DataKernel::FT

FT;

typedef typename DataKernel::Point_3

Point_3;

typedef typename DataKernel::Vector_3

Vector_3;

Creation

Monge_form monge_form; default constructor.

Access Functions

Point_3	*monge_form.origin()*	Point on the fitted surface where differential quantities are computed.

The Monge basis is given by:

Vector_3	*monge_form.maximal_principal_direction()*
Vector_3	*monge_form.minimal_principal_direction()*
Vector_3	*monge_form.normal_direction()*

The Monge coefficients are given by:

FT *monge_form.principal_curvatures(size_t i)*

$i = 0$ for the maximum and $i = 1$ for the minimum.

FT *monge_form.third_order_coefficients(size_t i)*

$0 \leq i \leq 3$

FT *monge_form.fourth_order_coefficients(size_t i)*

$0 \leq i \leq 4$

Operations

void *monge_form.comply_wrt_given_normal(const Vector_3 given_normal)*

change principal basis and Monge coefficients so that the given_normal and the Monge normal make an acute angle. If given_normal.monge_normal < 0 then change the orientation: if $z = g(x,y)$ in the basis (d1,d2,n) then in the basis (d2,d1,-n) $z = h(x,y) = -g(y,x)$.

The usual insert operator (*operator*<<) is overloaded for *Monge_form*, it gives the Monge coordinate system (the origin and an orthonormal basis) and the coefficients of the Monge form in this system.

See Also

Monge_via_jet_fitting

CGAL::Monge_via_jet_fitting<DataKernel, LocalKernel, SvdTraits>

Definition

The class *Monge_via_jet_fitting<DataKernel, LocalKernel, SvdTraits>* is designed to perform the estimation of the local differential quantities at a given point. The point range is given by a pair of input iterators, and it is assumed that the point where the calculation is carried out is the point that the begin iterator refers to. The results are stored in an instance of the nested class *Monge_form*, the particular information returned depending on the degrees specified for the polynomial fitting and for the Monge form.

The default for the template *LocalKernel* is *Cartesian<double>* and the default for *SvdTraits* is *Eigen_svd* if *CGAL_EIGEN3_ENABLED* is defined.

#include <CGAL/Monge_via_jet_fitting.h>

Parameters

The class *Monge_via_jet_fitting<DataKernel, LocalKernel, SvdTraits>* has three template parameters. Parameter *DataKernel* provides the geometric classes and tools corresponding to the input points, and also members of the *Monge_form* class. Parameter *LocalKernel* provides the geometric classes and tools required by local computations. Parameter *SvdTraits* features the linear algebra algorithm required by the fitting method.

Types

typedef DataKernel *Data_kernel;*
typedef LocalKernel *Local_kernel;*
typedef typename Local_kernel::FT

FT;
typedef typename Local_kernel::Vector_3

Vector_3;

Monge_via_jet_fitting<DataKernel, LocalKernel, SvdTraits>:: Monge_form

see the section below.

Creation

Monge_via_jet_fitting<DataKernel, LocalKernel, SvdTraits> *monge_fitting*;

default constructor

Operations

template <class InputIterator>

Monge_form *monge_fitting(InputIterator begin, InputIterator end, size_t d, size_t d')*

This operator performs all the computations. The N input points are given by the *InputIterator* parameters which value-type are *Data_kernel::Point_3*, d is the degree of the fitted polynomial, *d'* is the degree of the expected Monge coefficients.
Precondition: $N \geq N_d := (d+1)(d+2)/2$, $1 \leq d' \leq \min(d,4)$.

FT *monge_fitting.condition_number()*

condition number of the linear fitting system.

std::pair<FT, Vector_3>

monge_fitting.pca_basis(size_t i)

pca eigenvalues and eigenvectors, the pca_basis has always 3 such pairs. Precondition : i ranges from 0 to 2.

See Also

Eigen_svd, *Monge_form*

SvdTraits

Definition

The concept SvdTraits describes the set of requirements to be fulfilled by any class used to instantiate the third template parameter of the class *Monge_via_jet_fitting<DataKernel,LocalKernel,SvdTraits>*.

It describes the linear algebra types and algorithms needed by the class *Monge_via_jet_fitting*.

Requirements

The scalar type, *SvdTraits::FT*, must be the same as that of the *LocalKernel* concept : *LocalKernel::FT*.

Types

SvdTraits:: FT	The scalar type.
SvdTraits:: Vector	The vector type.
SvdTraits:: Matrix	The matrix type.

Operations

SvdTraits vector(size_t n);	initialize all the elements of the vector to zero.

The type *Vector* has the access methods

size_t	*vector.size()*	
FT	*vector(size_t i)*	return the i^{th} entry, i from 0 to $size() - 1$.
void	*vector.set(size_t i, const FT value)*	set the i^{th} entry to *value*.
*FT**	*vector.vector()*	return the vector as an array.

The type *Matrix* has the access methods

SvdTraits matrix(size_t n1, size_t n2);	initialize all the entries of the matrix to zero.

size_t	*matrix.number_of_rows()*	
size_t	*matrix.number_of_columns()*	
FT	*matrix(size_t i, size_t j)*	return the entry at row i and column j, i from 0 to *number_of_rows - 1*, j from 0 to *number_of_columns - 1*.
void	*matrix.set(size_t i, size_t j, const FT value)*	set the entry at row i and column j to *value*.

The concept *SvdTraits* has a linear solver using a singular value decomposition algorithm.

FT *traits.solve(Matrix& M, Vector& B)*

Solves the system $MX = B$ (in the least square sense if M is not square) using a singular value decomposition and returns the condition number of M. The solution is stored in B.

Has Models

Eigen_svd,

See Also

LocalKernel

Chapter 58

Point Set Processing

Pierre Alliez, Laurent Saboret, Nader Salman

Contents

58.1 Introduction

This CGAL component implements methods to analyze and process unorganized 3D point sets. The input is an unorganized 3D point set, possibly with normal attributes (unoriented or oriented). This point set can be analyzed to measure geometric properties such as average spacing between the points and their K nearest neighbors. It can be processed with functions devoted to the simplification, outlier removal, smoothing, normal estimation and normal orientation. The processing of point sets is often needed in applications dealing with measurement data, such as surface reconstruction from laser scanned data (see Figure 58.1).

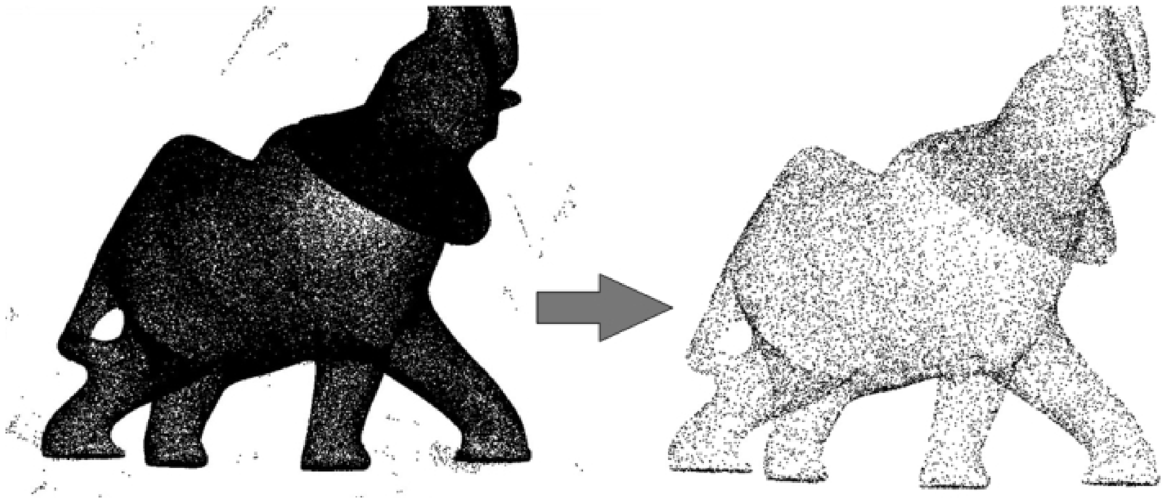


Figure 58.1: Point set processing. Left: 275K points sampled on the statue of an elephant with a Minolta laser scanner. Right: point set after clean-up and simplification to 17K points.

In the context of surface reconstruction we can position the elements of this component along the common surface reconstruction pipeline (Figure 58.2) which involves the following steps: 1) Scanning and scan alignment to produce a set of points or points with normals (alignment is not yet covered in CGAL); 2) Outlier removal; 3) Simplification to reduce the number of input points; 4) Smoothing to reduce noise in the input data; 5) Normal estimation and orientation when the normals are not already provided by the acquisition device; and 6) Surface reconstruction. Chapter *Surface_reconstruction_points_3* 49 deals with surface reconstruction from point sets with normal attributes.

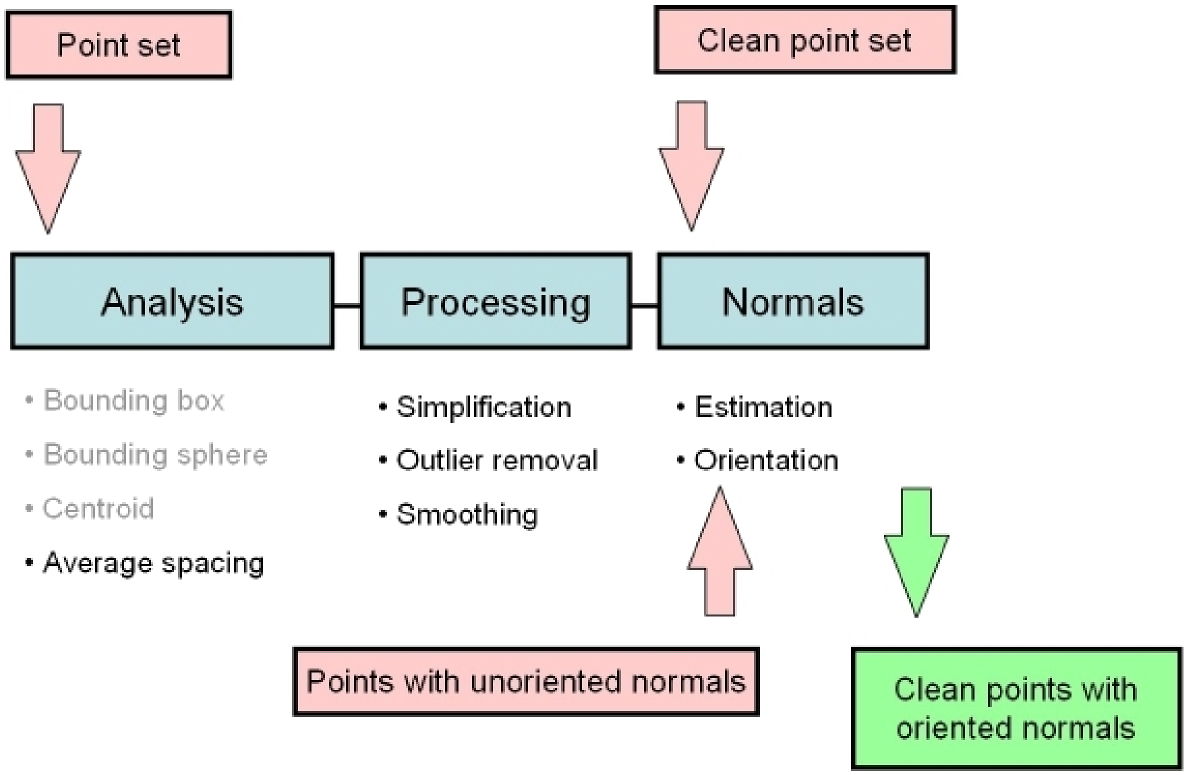


Figure 58.2: Point set processing pipeline for surface reconstruction. The algorithms listed in gray are available from other CGAL components (bounding volumes and principal component analysis).

58.2 Input/Output

58.2.1 Property Maps

The algorithms of this component take as input parameters iterator ranges of 3D points, or of 3D points with normals. The property maps are used to access the point or normal information from the input data, so as to let the user decide upon the implementation of a point with normal. The latter can be represented as, e.g., a class derived from the CGAL 3D point, or as a *std::pair<Point_3<K>, Vector_3<K> >*, or as a *boost::tuple<..,Point_3<K>, ..., Vector_3<K> >*.

The following classes described in Chapter 75 provide property maps for the implementations of points with normals listed above:
CGAL::Dereference_property_map<T>
CGAL::First_of_pair_property_map<Pair> and *CGAL::Second_of_pair_property_map<Pair>*
CGAL::Nth_of_tuple_property_map<N, Tuple>

Dereference_property_map<Point_3> is the default value of the position property map expected by all functions in this component.
See below examples using pair and tuple property maps.
Users of this package may use other types to represent positions and normals if they implement the corresponding property maps.

58.2.2 Streams

We provide functions to read and write sets of points or sets of points with normals from the following ASCII file formats: XYZ (three point coordinates *x y z* per line or three point coordinates and three normal vector coordinates *x y z nx ny nz* per line), and OFF (Object File Format) [Phi96].

CGAL::read_xyz_points
CGAL::read_off_points
CGAL::write_off_points
CGAL::write_xyz_points

58.2.3 Example

The following example reads a point set from an input file and writes it to a file, both in the XYZ format. Positions and normals are stored in pairs and accessed through property maps.

```
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/property_map.h>
#include <CGAL/IO/read_xyz_points.h>
#include <CGAL/IO/write_xyz_points.h>

#include <utility> // defines std::pair
#include <vector>
#include <fstream>

// types
typedef CGAL::Exact_predicates_inexact_constructions_kernel Kernel;
typedef Kernel::Point_3 Point;
typedef Kernel::Vector_3 Vector;

// Point with normal vector stored as a std::pair.
typedef std::pair<Point, Vector> Pwn;

int main(void)
{
    // Reads a .xyz point set file in points[].
    // Note: read_xyz_points_and_normals() requires an output iterator
    // over points and as well as property maps to access each
    // point position and normal.
    std::vector<Pwn> points;
    std::ifstream in("data/oni.xyz");
    if (!in ||
        !CGAL::read_xyz_points_and_normals(
            in,std::back_inserter(points),
            CGAL::First_of_pair_property_map<Pwn>(),
            CGAL::Second_of_pair_property_map<Pwn>()))
    {
      std::cerr << "Error: cannot read file data/oni.xyz" << std::endl;
      return EXIT_FAILURE;
    }

    // Saves point set.
```

```
    // Note: write_xyz_points_and_normals() requires an output iterator
    // over points as well as property maps to access each
    // point position and normal.
    std::ofstream out("oni_copy.xyz");
    if (!out ||
        !CGAL::write_xyz_points_and_normals(
            out, points.begin(), points.end(),
            CGAL::First_of_pair_property_map<Pwn>(),
            CGAL::Second_of_pair_property_map<Pwn>()))
    {
      return EXIT_FAILURE;
    }

    return EXIT_SUCCESS;
}
```

File: examples/Point_set_processing_3/read_write_xyz_point_set_example.cpp

58.3 Analysis

Function *CGAL::compute_average_spacing()* computes the average spacing of all input points to their k nearest neighbor points, k being specified by the user. As it provides an order of a point set density, this function is used downstream the surface reconstruction pipeline to automatically determine some parameters such as output mesh sizing for surface reconstruction.

Example

The following example reads a point set in the *xyz* format and computes the average spacing. Index, position and color are stored in a tuple and accessed through property maps.

```
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/compute_average_spacing.h>
#include <CGAL/IO/read_xyz_points.h>

#include <vector>
#include <fstream>
#include <boost/tuple/tuple.hpp>

// Types
typedef CGAL::Exact_predicates_inexact_constructions_kernel Kernel;
typedef Kernel::FT FT;
typedef Kernel::Point_3 Point;

// Data type := index, followed by the point, followed by three integers
    that
// define the Red Green Blue color of the point.
typedef boost::tuple<int, Point, int, int, int>
    IndexedPointWithColorTuple;
```

```
int main(void)
{
    // Reads a .xyz point set file in points.
    // As the point is the second element of the tuple (that is with
   index 1)
    // we use a property map that accesses the 1st element of the tuple.
    std::vector<IndexedPointWithColorTuple> points;
    std::ifstream stream("data/sphere_20k.xyz");
    if (!stream ||
        !CGAL::read_xyz_points(
            stream, std::back_inserter(points),

   CGAL::Nth_of_tuple_property_map<1,IndexedPointWithColorTuple>()))
    {
      std::cerr << "Error: cannot read file data/sphere_20k.xyz" <<
   std::endl;
      return EXIT_FAILURE;
    }

    // Initialize index and RGB color fields in tuple.
    // As the index and RGB color are respectively the first and
   third-fifth elements
    // of the tuple we use a get function from the property map that
   accesses the 0
    // and 2-4th elements of the tuple.
    for(unsigned int i = 0; i < points.size(); i++)
    {
      points[i].get<0>() = i;   // set index value of tuple to i

      points[i].get<2>() = 0;   // set RGB color to black
      points[i].get<3>() = 0;
      points[i].get<4>() = 0;
    }

    // Computes average spacing.
    const unsigned int nb_neighbors = 6; // 1 ring
    FT average_spacing = CGAL::compute_average_spacing(
                          points.begin(), points.end(),

   CGAL::Nth_of_tuple_property_map<1,IndexedPointWithColorTuple>(),
                          nb_neighbors);
    std::cout << "Average spacing: " << average_spacing << std::endl;

    return EXIT_SUCCESS;
}
```

File: examples/Point_set_processing_3/average_spacing_example.cpp

Note that other functions such as centroid or bounding volumes are found in other CGAL components:
CGAL::centroid
CGAL::bounding_box
Min_sphere_of_spheres_d<Traits>

58.4 Outlier Removal

Function *CGAL::remove_outliers()* deletes a user-specified fraction of outliers from an input point set. More specifically, it sorts the input points in increasing order of average squared distances to their k nearest neighbors and deletes the points with largest value.

Example

The following example reads a point set and removes 5% of the points. It uses the *CGAL::Dereference_property_map<Point_3>* property map (optional as it is the default position property map of all functions in this component.)

```
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/property_map.h>
#include <CGAL/remove_outliers.h>
#include <CGAL/IO/read_xyz_points.h>

#include <vector>
#include <fstream>

// types
typedef CGAL::Exact_predicates_inexact_constructions_kernel Kernel;
typedef Kernel::Point_3 Point;

int main(void)
{
  // Reads a .xyz point set file in points[].
  // The Dereference_property_map property map can be omitted here as it
   is the default value.
  std::vector<Point> points;
  std::ifstream stream("data/oni.xyz");
  if (!stream ||
      !CGAL::read_xyz_points(stream, std::back_inserter(points),
                             CGAL::Dereference_property_map<Point>()))
  {
    std::cerr << "Error: cannot read file data/oni.xyz" << std::endl;
    return EXIT_FAILURE;
  }

  // Removes outliers using erase-remove idiom.
  // The Dereference_property_map property map can be omitted here as it
   is the default value.
  const double removed_percentage = 5.0; // percentage of points to
   remove
  const int nb_neighbors = 24; // considers 24 nearest neighbor points
  points.erase(CGAL::remove_outliers(points.begin(), points.end(),

   CGAL::Dereference_property_map<Point>(),
                                     nb_neighbors, removed_percentage),
              points.end());

  // Optional: after erase(), use Scott Meyer's "swap trick" to trim
   excess capacity
```

```
  std::vector<Point>(points).swap(points);

  return EXIT_SUCCESS;
}
```

File: examples/Point_set_processing_3/remove_outliers_example.cpp

58.5 Simplification

Two simplification functions are devised to reduce an input point set, either randomly or using a grid-based clustering approach.

Function *CGAL::random_simplify_point_set()* randomly deletes a user-specified fraction of points from the input point set. This algorithm is fast.

Function *CGAL::grid_simplify_point_set()* considers a regular grid covering the bounding box of the input point set, and clusters all points sharing the same cell of the grid by picking as representant one arbitrarily chosen point. This algorithm is slower than *CGAL::random_simplify_point_set()*.

Example

The following example reads a point set and simplifies it by clustering.

```
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/grid_simplify_point_set.h>
#include <CGAL/IO/read_xyz_points.h>

#include <vector>
#include <fstream>

// types
typedef CGAL::Exact_predicates_inexact_constructions_kernel Kernel;
typedef Kernel::Point_3 Point;

int main(void)
{
  // Reads a .xyz point set file in points[].
  std::vector<Point> points;
  std::ifstream stream("data/oni.xyz");
  if (!stream ||
      !CGAL::read_xyz_points(stream, std::back_inserter(points)))
  {
    std::cerr << "Error: cannot read file data/oni.xyz" << std::endl;
    return EXIT_FAILURE;
  }

  // simplification by clustering using erase-remove idiom
  double cell_size = 0.001;
  points.erase(CGAL::grid_simplify_point_set(points.begin(),
   points.end(), cell_size),
```

```
              points.end());

  // Optional: after erase(), use Scott Meyer's "swap trick" to trim
    excess capacity
  std::vector<Point>(points).swap(points);

  return EXIT_SUCCESS;
}
```

File: examples/Point_set_processing_3/grid_simplification_example.cpp

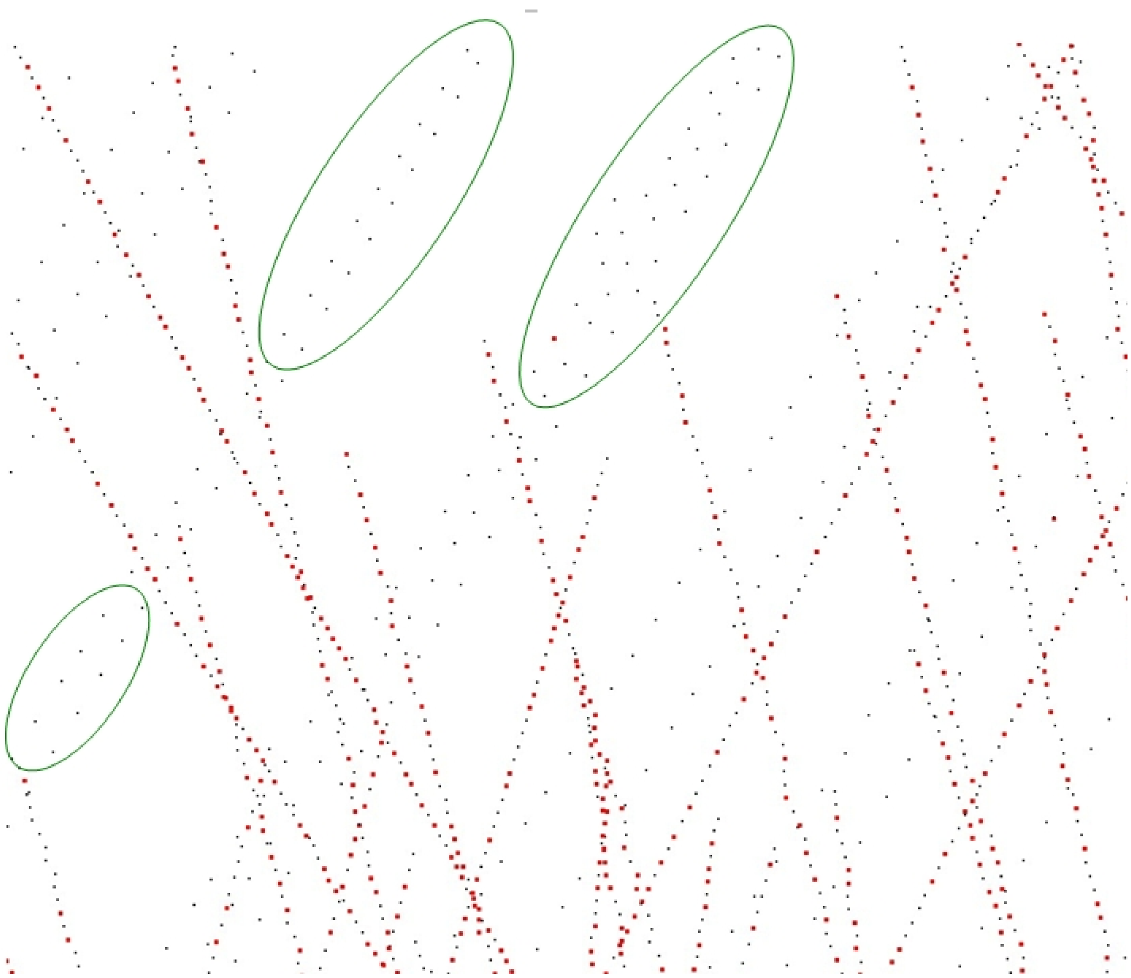


Figure 58.3: Point set simplification through grid-based clustering. Removed points are depicted in red. Notice how low-density areas (in green) are not simplified.

58.6 Smoothing

Function *CGAL::jet_smooth_point_set* smooths the input point set by projecting each point onto a smooth parametric surface patch (so-called jet surface) fitted over its k nearest neighbors.

Example

The following example generates a set of 9 points close to the xy plane and smooths them using 8 nearest neighbors:

```
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/jet_smooth_point_set.h>
#include <vector>

// types
typedef CGAL::Exact_predicates_inexact_constructions_kernel Kernel;
typedef Kernel::Point_3 Point;

int main(void)
{
  // generate point set
  std::vector<Point> points;
  points.push_back(Point( 0.0, 0.0, 0.001));
  points.push_back(Point(-0.1,-0.1, 0.002));
  points.push_back(Point(-0.1,-0.2, 0.001));
  points.push_back(Point(-0.1, 0.1, 0.002));
  points.push_back(Point( 0.1,-0.1, 0.000));
  points.push_back(Point( 0.1, 0.2, 0.001));
  points.push_back(Point( 0.2, 0.0, 0.002));
  points.push_back(Point( 0.2, 0.1, 0.000));
  points.push_back(Point( 0.0,-0.1, 0.001));

  // Smoothing.
  const unsigned int nb_neighbors = 8; // default is 24 for real-life
    point sets
  CGAL::jet_smooth_point_set(points.begin(), points.end(), nb_neighbors);

  return EXIT_SUCCESS;
}
```

File: examples/Point_set_processing_3/jet_smoothing_example.cpp

58.7 Normal Estimation

Assuming a point set sampled over an inferred surface S, two functions provide an estimate of the normal to S at each point. The result is an unoriented normal vector for each input point.

Function *CGAL::jet_estimate_normals()* estimates the normal direction at each point from the input set by fitting a jet surface over its k nearest neighbors. The default jet is a quadric surface. This algorithm is well suited to point sets scattered over curved surfaces.

Function *CGAL::pca_estimate_normals()* estimates the normal direction at each point from the set by linear least squares fitting of a plane over its k nearest neighbors. This algorithm is simpler and faster than *CGAL::jet_estimate_normals()*.

58.8 Normal Orientation

Function *CGAL::mst_orient_normals()* orients the normals of a set of points with unoriented normals using the method described by Hoppe et al. in *Surface reconstruction from unorganized points* [HDD+92]. More specifically, this method constructs a Riemannian graph over the input points (the graph of the K nearest neighbor points) and propagates a seed normal orientation within a minimum spanning tree computed over this graph. The result is an oriented normal vector for each input unoriented normal, except for the normals which could not be successfully oriented.

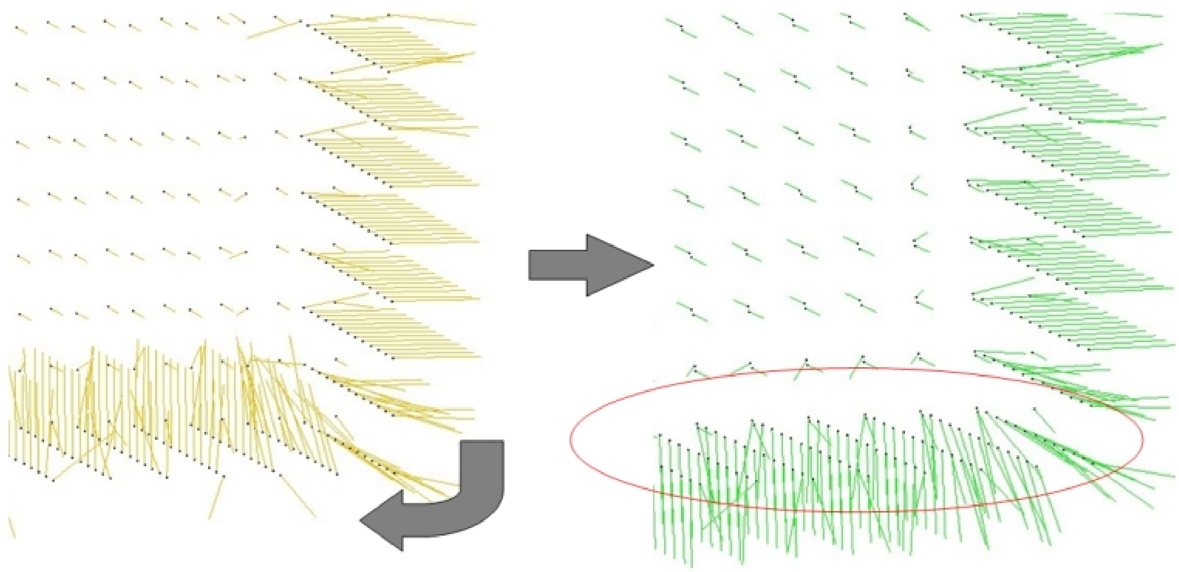


Figure 58.4: Normal orientation of a sampled cube surface. Left: unoriented normals. Right: orientation of right face normals is propagated to bottom face.

Example

The following example reads a point set from a file, estimates the normals through PCA over the 6 nearest neighbors and orients the normals:

```
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/pca_estimate_normals.h>
#include <CGAL/mst_orient_normals.h>
#include <CGAL/property_map.h>
#include <CGAL/IO/read_xyz_points.h>

#include <utility> // defines std::pair
```

```
#include <list>
#include <fstream>

// Types
typedef CGAL::Exact_predicates_inexact_constructions_kernel Kernel;
typedef Kernel::Point_3 Point;
typedef Kernel::Vector_3 Vector;

// Point with normal vector stored in a std::pair.
typedef std::pair<Point, Vector> PointVectorPair;

int main(void)
{
    // Reads a .xyz point set file in points[].
    std::list<PointVectorPair> points;
    std::ifstream stream("data/sphere_20k.xyz");
    if (!stream ||
        !CGAL::read_xyz_points(stream,
                               std::back_inserter(points),

   CGAL::First_of_pair_property_map<PointVectorPair>()))
    {
        std::cerr << "Error: cannot read file data/sphere_20k.xyz" <<
   std::endl;
        return EXIT_FAILURE;
    }

    // Estimates normals direction.
    // Note: pca_estimate_normals() requires an iterator over points
    // as well as property maps to access each point's position and
   normal.
    const int nb_neighbors = 18; // K-nearest neighbors = 3 rings
    CGAL::pca_estimate_normals(points.begin(), points.end(),

   CGAL::First_of_pair_property_map<PointVectorPair>(),

   CGAL::Second_of_pair_property_map<PointVectorPair>(),
                               nb_neighbors);

    // Orients normals.
    // Note: mst_orient_normals() requires an iterator over points
    // as well as property maps to access each point's position and
   normal.
    std::list<PointVectorPair>::iterator unoriented_points_begin =
        CGAL::mst_orient_normals(points.begin(), points.end(),

   CGAL::First_of_pair_property_map<PointVectorPair>(),

   CGAL::Second_of_pair_property_map<PointVectorPair>(),
                                 nb_neighbors);

    // Optional: delete points with an unoriented normal
    // if you plan to call a reconstruction algorithm that expects
   oriented normals.
```

```
    points.erase(unoriented_points_begin, points.end());

    // Optional: after erase(), use Scott Meyer's "swap trick" to trim
   excess capacity
    std::list<PointVectorPair>(points).swap(points);

    return EXIT_SUCCESS;
}
```

File: examples/Point_set_processing_3/normals_example.cpp

Point Set Processing Reference Manual

Pierre Alliez, Laurent Saboret, Nader Salman

This CGAL component implements methods to analyze and process unorganized point sets. The input is an unorganized point set, possibly with attributes. This point set can be analyzed to measure average spacing, a global estimation of the point set density. Furthermore we can process it with functions devoted to simplification, outlier removal, smoothing, normal estimation and normal orientation.

58.9 Classified Reference Pages

58.9.1 Functions

58.10 Alphabetical List of Reference Pages

CGAL::compute_average_spacing

Function

Definition

CGAL::compute_average_spacing() computes the average spacing of all points from the input set to their k nearest neighbors. This value depends on the k parameter which can be set to 6 for isotropically sampled surfaces.

#include <CGAL/compute_average_spacing.h>

```
template<typename InputIterator, typename PointPMap, typename Kernel>
Kernel::FT      compute_average_spacing( InputIterator first,
                                         InputIterator beyond,
                                         PointPMap point_pmap,
                                         unsigned int k,
                                         Kernel kernel)
```

Computes average spacing from k nearest neighbors.
Precondition: k >= 2.
Template Parameters:
InputIterator: iterator over input points. *PointPMap*: is a model of *boost::ReadablePropertyMap* with a *value_type = Point_3<Kernel>*. It can be omitted if *InputIterator value_type* is convertible to *Point_3<Kernel>*. *Kernel*: Geometric traits class. It can be omitted and deduced automatically from *PointPMap value_type*.
Returns: average spacing (scalar).
Parameters:
first: iterator over the first input point. *beyond*: past-the-end iterator over the input points. *point_pmap*: property map *InputIterator* -> *Point_3*. *k*: number of neighbors. *kernel*: geometric traits.

Example

See *average_spacing_example.cpp*.

Function

CGAL::grid_simplify_point_set

Definition

Function *CGAL::grid_simplify_point_set()* considers a regular grid covering the bounding box of the input point set, and clusters all points sharing the same cell of the grid by picking as representant one arbitrarily chosen point.

This method modifies the order of input points so as to pack all remaining points first, and returns an iterator over the first point to remove (see erase-remove idiom). For this reason it should not be called on sorted containers.

#include <CGAL/grid_simplify_point_set.h>

```
template<typename ForwardIterator, typename PointPMap, typename Kernel>
ForwardIterator        grid_simplify_point_set( ForwardIterator first,
                                                ForwardIterator beyond,
                                                PointPMap point_pmap,
                                                double epsilon,
                                                Kernel kernel)
```

Merges points which belong to the same cell of a grid of cell size = epsilon. This method modifies the order of input points so as to pack all remaining points first, and returns an iterator over the first point to remove (see erase-remove idiom). For this reason it should not be called on sorted containers.
Precondition: epsilon > 0.
Template Parameters:
ForwardIterator: iterator over input points. *PointPMap*: is a model of *boost::ReadablePropertyMap* with a *value_type* = *Point_3<Kernel>*. It can be omitted if *ForwardIterator value_type* is convertible to *Point_3<Kernel>*. *Kernel*: Geometric traits class. It can be omitted and deduced automatically from *PointPMap value_type*.
Returns: iterator over the first point to remove.
Parameters:
first: iterator over the first input point. *beyond*: past-the-end iterator over the input points. *point_pmap*: property map *ForwardIterator* -> *Point_3*. *epsilon*: tolerance value when merging 3D points. *kernel*: geometric traits.

See Also

CGAL::random_simplify_point_set . . . . page 3512

Example

See *grid_simplification_example.cpp*.

CGAL::jet_estimate_normals

Definition

CGAL::jet_estimate_normals() estimates normal directions of all points from the input set by fitting jet surfaces over the k nearest neighbors. The default jet surface is a quadric, and the result is an unoriented normal vector for each input point.

The output of the normal estimation algorithm highly depends on the k parameter. The number of neighbors controls the size of the point subset considered for jet fitting at each input point. As this parameter is application-specific we do not provide any default value. Larger values lead to smoother normal fields and are more time consuming. For point sets with limited noise this value can be set to small number such as 18. For noisy point sets this value must be increased.

#include <CGAL/jet_estimate_normals.h>

```
template<typename InputIterator, typename PointPMap, typename NormalPMap, typename Kernel>
void    jet_estimate_normals( InputIterator first,
                              InputIterator beyond,
                              PointPMap point_pmap,
                              NormalPMap normal_pmap,
                              unsigned int k,
                              Kernel kernel,
                              unsigned int degree_fitting = 2)
```

Estimates normal directions of the [first, beyond) range of points using jet fitting on the k nearest neighbors. The output normals are randomly oriented.
Precondition: k >= 2.
Template Parameters:
InputIterator: iterator over input points. *PointPMap*: is a model of *boost::ReadablePropertyMap* with a *value_type = Point_3<Kernel>*. It can be omitted if *InputIterator value_type* is convertible to *Point_3<Kernel>*. *NormalPMap*: is a model of *boost::WritablePropertyMap* with a *value_type = Vector_3<Kernel>*. *Kernel*: Geometric traits class. It can be omitted and deduced automatically from *PointPMap value_type*.
Parameters:
first: iterator over the first input point. *beyond*: past-the-end iterator over the input points. *point_pmap*: property map *InputIterator -> Point_3*. *normal_pmap*: property map *InputIterator -> Vector_3*. *k*: number of neighbors. *kernel*: geometric traits.

See Also

CGAL::pca_estimate_normals . . . . page 3511
CGAL::mst_orient_normals . . . . page 3509

Example

See *normal_estimation.cpp* example.

Function

CGAL::jet_smooth_point_set

Definition

CGAL::jet_smooth_point_set() smooths a point set by fitting for each point a jet surface and projecting it onto the jet. The default jet surface is a quadric.

The output of the smoothing algorithm highly depends on the *k* parameter. The number of neighbors controls the size of the point subset considered for jet fitting at each input point. As this parameter is application-specific we do not provide any default value. Larger values lead to smoother point sets and are more time consuming. For point sets with limited noise this value can be set to small number such as 24. For noisy point sets this value must be increased.

#include <CGAL/jet_smooth_point_set.h>

```
template<typename InputIterator, typename PointPMap, typename Kernel>
void    jet_smooth_point_set( InputIterator first,
                              InputIterator beyond,
                              PointPMap point_pmap,
                              unsigned int k,
                              Kernel kernel,
                              unsigned int degree_fitting = 2,
                              unsigned int degree_monge = 2)
```

Smoothes the [first, beyond) range of points using jet fitting on the k nearest neighbors and reprojection onto the jet. As this method relocates the points, it should not be called on containers sorted w.r.t. point locations.
Precondition: k >= 2.
Template Parameters:
InputIterator: iterator over input points. *PointPMap*: is a model of *boost::ReadablePropertyMap* with a *value_type = Point_3<Kernel>*. It can be omitted if *InputIterator value_type* is convertible to *Point_3<Kernel>*. *Kernel*: Geometric traits class. It can be omitted and deduced automatically from *PointPMap value_type*.
Parameters:
first: iterator over the first input point. *beyond*: past-the-end iterator over the input points. *point_pmap*: property map *InputIterator -> Point_3*. *k*: number of neighbors. *kernel*: geometric traits.

Example

See *jet_smoothing_example.cpp*.

CGAL::mst_orient_normals

Function

Definition

CGAL::mst_orient_normals() orients the normals of a point set using the propagation of a seed orientation through a minimum spanning tree computed over the Riemannian graph [HDD+92].

The seed is chosen as the top point of the point set. Its normal is oriented towards +Z axis. The success of the orientation algorithm depends on the *k* parameter. The number of neighbors controls the number of candidates to propagate the orientation to around each input point. In general the value 18 works well. With smaller values the propagation may be blocked by large gaps in sparse point sets as the graph may be disconnected. Large values cause problems with points scattered over thin objects as the algorithm may incorrectly propagate the orientation from one side of the object to the other. In presence of disconnected clusters of points the algorithm may fail propagating the orientation from one cluster to the others and may only orient the top cluster.

This method modifies the order of the input points so as to pack all successfully oriented normals first, and returns an iterator over the first point with an unoriented normal (see erase-remove idiom). For this reason it should not be called on sorted containers.

#include <CGAL/mst_orient_normals.h>

```
template<typename ForwardIterator, typename PointPMap, typename NormalPMap, typename Kernel>
ForwardIterator    mst_orient_normals( ForwardIterator first,
                                       ForwardIterator beyond,
                                       PointPMap point_pmap,
                                       NormalPMap normal_pmap,
                                       unsigned int k,
                                       Kernel kernel)
```

Orients the normals of the [first, beyond) range of points using the propagation of a seed orientation through a minimum spanning tree of the Riemannian graph [HDD+92].
This method modifies the order of input points so as to pack all sucessfully oriented points first, and returns an iterator over the first point with an unoriented normal (see erase-remove idiom). For this reason it should not be called on sorted containers.
Preconditions:
Normals must be unit vectors. k >= 2.
Template Parameters:
ForwardIterator: iterator over input points. *PointPMap*: is a model of *boost::ReadablePropertyMap* with a *value_type* = *Point_3<Kernel>*. It can be omitted if *ForwardIterator value_type* is convertible to *Point_3<Kernel>*. *NormalPMap*: is a model of *boost::ReadWritePropertyMap* with a *value_type* = *Vector_3<Kernel>*. *Kernel*: Geometric traits class. It can be omitted and deduced automatically from *PointPMap value_type*.
Returns: iterator over the first point with an unoriented normal.
Parameters:
first: iterator over the first input point. *beyond*: past-the-end iterator over the input points. *point_pmap*: property map *ForwardIterator -> Point_3*. *normal_pmap*: property map *ForwardIterator -> Vector_3*. *k*: number of neighbors. *kernel*: geometric traits.

See Also

CGAL::pca_estimate_normals .... page 3511
CGAL::jet_estimate_normals .... page 3507

Example

See *pca_estimate_normals_example.cpp*.

CGAL::pca_estimate_normals

Function

Definition

CGAL::pca_estimate_normals() estimates normal directions at all points of an input point set by linear least squares fitting of a plane over their *k* nearest neighbors. The result is an unoriented normal for each input point.

The output of the normal estimation algorithm highly depends on the *k* parameter. The number of neighbors controls the size of the point subset considered for plane fitting at each input point. As this parameter is application-specific we do not provide any default value. For noise-free point sets this value can be set to a small number, e.g., 18. Larger values (e.g., 30 or more) lead to smoother normal fields and are more time consuming. We thus recommend using them only for noisy data sets.

#include <CGAL/pca_estimate_normals.h>

```
template<typename InputIterator, typename PointPMap, typename NormalPMap, typename Kernel>
void    pca_estimate_normals( InputIterator first,
                              InputIterator beyond,
                              PointPMap point_pmap,
                              NormalPMap normal_pmap,
                              unsigned int k,
                              Kernel kernel)
```

Estimates normal directions of the [first, beyond) range of points by linear least squares fitting of a plane over the k nearest neighbors. The output normals are randomly oriented.
Precondition: k >= 2.
Template Parameters:
InputIterator: iterator over input points. *PointPMap*: is a model of *boost::ReadablePropertyMap* with a *value_type = Point_3<Kernel>*. It can be omitted if *InputIterator value_type* is convertible to *Point_3<Kernel>*. *NormalPMap*: is a model of *boost::WritablePropertyMap* with a *value_type = Vector_3<Kernel>*. *Kernel*: Geometric traits class. It can be omitted and deduced automatically from *PointPMap value_type*.
Parameters:
first: iterator over the first input point. *beyond*: past-the-end iterator over the input points. *point_pmap*: property map *InputIterator -> Point_3*. *normal_pmap*: property map *InputIterator -> Vector_3*. *k*: number of neighbors. *kernel*: geometric traits.

See Also

CGAL::jet_estimate_normals .... page 3507
CGAL::mst_orient_normals .... page 3509

Example

See *normals_example.cpp*.

Function

CGAL::random_simplify_point_set

Definition

CGAL::random_simplify_point_set() randomly deletes a user-specified fraction of the input points. This method modifies the order of input points so as to pack all remaining points first, and returns an iterator over the first point to remove (see erase-remove idiom). For this reason it should not be called on sorted containers.

#include <CGAL/random_simplify_point_set.h>

```
template<typename ForwardIterator, typename PointPMap, typename Kernel>
ForwardIterator    random_simplify_point_set( ForwardIterator first,
                                              ForwardIterator beyond,
                                              PointPMap point_pmap,
                                              double removed_percentage,
                                              Kernel kernel)
```

Randomly deletes a user-specified fraction of the input points.
This method modifies the order of input points so as to pack all remaining points first, and returns an iterator over the first point to remove (see erase-remove idiom). For this reason it should not be called on sorted containers.
Template Parameters:
ForwardIterator: iterator over input points. *PointPMap*: is a model of *boost::ReadablePropertyMap* with a *value_type = Point_3<Kernel>*. It can be omitted if *ForwardIterator value_type* is convertible to *Point_3<Kernel>*. *Kernel*: Geometric traits class. It can be omitted and deduced automatically from *PointPMap value_type*.
Returns: iterator over the first point to remove.
Parameters:
first: iterator over the first input point. *beyond*: past-the-end iterator over the input points. *point_pmap*: property map *ForwardIterator -> Point_3*. *removed_percentage*: percentage of points to remove. *kernel*: geometric traits.

See Also

CGAL::grid_simplify_point_set . . . . page 3506

Example

See *random_simplification_example.cpp*.

CGAL::read_off_points

Definition

CGAL::read_off_points() reads the point from a .off ASCII stream. More specifically, it reads only the point locations and ignores all point attributes available from the stream. *CGAL::read_off_points_and_normals()* reads the points as well as the normals (if available) from a .off ASCII stream. In both cases the other primitives (segments, faces) are ignored.

#include <CGAL/IO/read_off_points.h>

template<typename OutputIterator, typename PointPMap, typename Kernel>
bool read_off_points(std::istream& stream, OutputIterator output, PointPMap point_pmap, Kernel kernel)

Reads points (position only) from a .off ASCII stream. The function expects for each point a line with the x y z position. If the position is followed by the nx ny nz normal, then the normal will be ignored. Faces are ignored.
Template Parameters:
OutputIterator: iterator over output points. *PointPMap*: is a model of *boost::WritablePropertyMap* with a *value_type = Point_3<Kernel>*. It can be omitted if *OutputIterator value_type* is convertible to *Point_3<Kernel>*. *Kernel*: Geometric traits class. It can be omitted and deduced automatically from *PointPMap value_type*.
Returns: true on success.
Parameters:
stream: input stream. *output*: output iterator over points. *point_pmap*: property map *OutputIterator -> Point_3*. *kernel*: geometric traits.

template<typename OutputIterator, typename PointPMap, typename NormalPMap, typename Kernel>
bool read_off_points_and_normals(std::istream& stream, OutputIterator output, PointPMap point_pmap, NormalPMap normal_pmap, Kernel kernel)

Reads points (positions + normals, if available) from a .off ASCII stream. The function expects for each point a line with the x y z position, optionally followed by the nx ny nz normal. Faces are ignored.
Template Parameters:
OutputIterator: iterator over output points. *PointPMap*: is a model of *boost::WritablePropertyMap* with a *value_type = Point_3<Kernel>*. It can be omitted if *OutputIterator value_type* is convertible to *Point_3<Kernel>*. *NormalPMap*: is a model of *boost::WritablePropertyMap* with a *value_type = Vector_3<Kernel>*. *Kernel*: Geometric traits class. It can be omitted and deduced automatically from *PointPMap value_type*.
Returns: true on success.
Parameters:
stream: input stream. *output*: output iterator over points. *point_pmap*: property map *OutputIterator -> Point_3*. *normal_pmap*: property map *OutputIterator -> Vector_3*. *kernel*: geometric traits.

See Also

Example

See *read_write_xyz_point_set_example.cpp*.

CGAL::read_xyz_points

Function

Definition

CGAL::read_xyz_points() reads the point locations from a .xyz ASCII stream. *CGAL::read_xyz_points_and_normals()* reads the points as well as the normals (if available) from a .xyz ASCII stream.

#include <CGAL/IO/read_xyz_points.h>

template<typename OutputIterator, typename PointPMap, typename Kernel>
bool read_xyz_points(std::istream& stream, OutputIterator output, PointPMap point_pmap, Kernel kernel)

Reads points (positions only) from a .xyz ASCII stream. The function expects for each point a line with the x y z position. If the position is followed by the nx ny nz normal, then the normal will be ignored. The first line may contain the number of points in the file. Empty lines and comments starting by # character are allowed.
Template Parameters:
OutputIterator: iterator over output points. *PointPMap*: is a model of *boost::WritablePropertyMap* with a *value_type = Point_3<Kernel>*. It can be omitted if *OutputIterator value_type* is convertible to *Point_3<Kernel>*. *Kernel*: Geometric traits class. It can be omitted and deduced automatically from *PointPMap value_type*.
Returns: true on success.
Parameters:
stream: input stream. *output*: output iterator over points. *point_pmap*: property map *OutputIterator -> Point_3*. *kernel*: geometric traits.

template<typename OutputIterator, typename PointPMap, typename NormalPMap, typename Kernel>
bool read_xyz_points_and_normals(std::istream& stream, OutputIterator output, PointPMap point_pmap, NormalPMap normal_pmap,

Kernel kernel)

Reads points (positions + normals, if available) from a .xyz ASCII stream. The function expects for each point a line with the x y z position, optionally followed by the nx ny nz normal. The first line may contain the number of points in the file. Empty lines and comments starting by # character are allowed.

Template Parameters:

OutputIterator: iterator over output points. *PointPMap*: is a model of *boost::WritablePropertyMap* with a *value_type* = *Point_3<Kernel>*. It can be omitted if *OutputIterator value_type* is convertible to *Point_3<Kernel>*. *NormalPMap*: is a model of *boost::WritablePropertyMap* with a *value_type* = *Vector_3<Kernel>*. *Kernel*: Geometric traits class. It can be omitted and deduced automatically from *PointPMap value_type*.

Returns: true on success.

Parameters:

stream: input stream. *output*: output iterator over points. *point_pmap*: property map *OutputIterator* -> *Point_3*. *normal_pmap*: property map *OutputIterator* -> *Vector_3*. *kernel*: geometric traits.

See Also

CGAL::write_xyz_points . . . . page 3520
CGAL::read_off_points . . . . page 3513
CGAL::write_off_points . . . . page 3518

Example

See *read_write_xyz_point_set_example.cpp*.

CGAL::remove_outliers

Definition

CGAL::remove_outliers() deletes a user-specified fraction of outliers from the input point set. More specifically, it sorts the input points in increasing order of average squared distances to the k nearest neighbors and computes the points with largest value.

The outliers detection depends on the k parameter, specifically the detection of clusters of outliers. The number of neighbors should be higher than the size of clusters of outliers in the point set. For datasets with no cluster of outliers, this value can be set to a few rings, e.g. 24. Larger value leads to longer computation times. For these reasons, we do not provide any default value for this parameter.

This method modifies the order of input points so as to pack all remaining points first, and returns and returns an iterator over the first point to remove (see erase-remove idiom). For this reason it should not be called on sorted containers.

#include <CGAL/remove_outliers.h>

template<typename InputIterator, typename PointPMap, typename Kernel>
InputIterator remove_outliers(InputIterator first,
InputIterator beyond,
PointPMap point_pmap,
unsigned int k,
double threshold_percent,
Kernel kernel)

Removes outliers: computes average squared distance to the K nearest neighbors, and sorts the points in increasing order of average distance.
This method modifies the order of input points so as to pack all remaining points first, and returns an iterator over the first point to remove (see erase-remove idiom). For this reason it should not be called on sorted containers.
Precondition: k >= 2.
Template Parameters:
InputIterator: iterator over input points. *PointPMap*: is a model of *boost::ReadablePropertyMap* with a *value_type = Point_3<Kernel>*. It can be omitted if *InputIterator value_type* is convertible to *Point_3<Kernel>*. *Kernel*: Geometric traits class. It can be omitted and deduced automatically from *PointPMap value_type*.
Returns: iterator over the first point to remove.
Parameters:
first: iterator over the first input point. *beyond*: past-the-end iterator over the input points. *point_pmap*: property map *InputIterator -> Point_3*. *k*: number of neighbors. *threshold_percent*: percentage of points to remove. *kernel*: geometric traits.

Example

See *remove_outliers_example.cpp*.

Function

CGAL::write_off_points

Definition

CGAL::write_off_points() saves the points of an iterator range to a .off ASCII stream. More specifically, it saves only the point locations and ignores all other attributes. *CGAL::write_off_points_and_normals()* saves the points as well the normals of an iterator range to a .off ASCII stream.

#include <CGAL/IO/write_off_points.h>

```
template<typename ForwardIterator, typename PointPMap, typename Kernel>
bool          write_off_points( std::ostream& stream,
                                ForwardIterator first,
                                ForwardIterator beyond,
                                PointPMap point_pmap,
                                Kernel kernel)
```

Saves the [first, beyond) range of points (positions only) to a .off ASCII stream. The function writes for each point a line with the x y z position.
Template Parameters:
ForwardIterator: iterator over input points. *PointPMap*: is a model of *boost::ReadablePropertyMap* with a *value_type* = *Point_3<Kernel>*. It can be omitted if *ForwardIterator value_type* is convertible to *Point_3<Kernel>*. *Kernel*: Geometric traits class. It can be omitted and deduced automatically from *PointPMap value_type*.
Returns: true on success.
Parameters:
stream: output stream. *first*: iterator over the first input point. *beyond*: past-the-end iterator over the input points. *point_pmap*: property map *ForwardIterator* -> *Point_3*. *kernel*: geometric traits.

```
template<typename ForwardIterator, typename PointPMap, typename NormalPMap, typename Kernel>
bool          write_off_points_and_normals( std::ostream& stream,
                                            ForwardIterator first,
                                            ForwardIterator beyond,
                                            PointPMap point_pmap,
                                            NormalPMap normal_pmap,
```

Kernel kernel)

Saves the [first, beyond) range of points (positions + normals) to a .off ASCII stream. The function writes for each point a line with the x y z position followed by the nx ny nz normal.
Precondition: normals must be unit vectors.
Template Parameters:
ForwardIterator: iterator over input points. *PointPMap*: is a model of *boost::ReadablePropertyMap* with a *value_type* = *Point_3*<*Kernel*>. It can be omitted if *ForwardIterator value_type* is convertible to *Point_3*<*Kernel*>. *NormalPMap*: is a model of *boost::WritablePropertyMap* with a *value_type* = *Vector_3*<*Kernel*>. *Kernel*: Geometric traits class. It can be omitted and deduced automatically from *PointPMap value_type*.
Returns: true on success.
Parameters:
stream: output stream. *first*: iterator over the first input point. *beyond*: past-the-end iterator over the input points. *point_pmap*: property map *ForwardIterator* -> *Point_3*. *normal_pmap*: property map *ForwardIterator* -> *Vector_3*. *kernel*: geometric traits.

See Also

CGAL::read_xyz_points ..... page 3515
CGAL::write_xyz_points ..... page 3520
CGAL::read_off_points ..... page 3513

Example

See *read_write_xyz_point_set_example.cpp*.

Function

CGAL::write_xyz_points

Definition

CGAL::write_xyz_points() saves points (positions only) to a .xyz ASCII stream. *CGAL::write_xyz_points_and_normals()* saves points (positions + normals) to a .xyz ASCII stream.

#include <CGAL/IO/write_xyz_points.h>

```
template<typename ForwardIterator, typename PointPMap, typename Kernel>
bool write_xyz_points( std::ostream& stream,
                       ForwardIterator first,
                       ForwardIterator beyond,
                       PointPMap point_pmap,
                       Kernel kernel)
```

Saves the [first, beyond) range of points (positions only) to a .xyz ASCII stream. The function writes for each point a line with the x y z position.
Template Parameters:
ForwardIterator: iterator over input points. *PointPMap*: is a model of *boost::ReadablePropertyMap* with a *value_type* = *Point_3<Kernel>*. It can be omitted if *ForwardIterator value_type* is convertible to *Point_3<Kernel>*. *Kernel*: Geometric traits class. It can be omitted and deduced automatically from *PointPMap value_type*.
Returns: true on success.
Parameters:
stream: output stream. *first*: iterator over the first input point. *beyond*: past-the-end iterator over the input points. *point_pmap*: property map *ForwardIterator -> Point_3*. *kernel*: geometric traits.

```
template<typename ForwardIterator, typename PointPMap, typename NormalPMap, typename Kernel>
bool write_xyz_points_and_normals( std::ostream& stream,
                                   ForwardIterator first,
                                   ForwardIterator beyond,
                                   PointPMap point_pmap,
                                   NormalPMap normal_pmap,
```

Kernel kernel)

Saves the [first, beyond) range of points (positions + normals) to a .xyz ASCII stream. The function writes for each point a line with the x y z position followed by the nx ny nz normal.
Precondition: normals must be unit vectors.
Template Parameters:
ForwardIterator: iterator over input points. *PointPMap*: is a model of *boost::ReadablePropertyMap* with a *value_type* = *Point_3*<*Kernel*>. It can be omitted if *ForwardIterator value_type* is convertible to *Point_3*<*Kernel*>. *NormalPMap*: is a model of *boost::WritablePropertyMap* with a *value_type* = *Vector_3*<*Kernel*>. *Kernel*: Geometric traits class. It can be omitted and deduced automatically from *PointPMap value_type*.
Returns: true on success.
Parameters:
stream: output stream. *first*: iterator over the first input point. *beyond*: past-the-end iterator over the input points. *point_pmap*: property map *ForwardIterator* -> *Point_3*. *normal_pmap*: property map *ForwardIterator* -> *Vector_3*. *kernel*: geometric traits.

See Also

CGAL::read_xyz_points . . . . page 3515
CGAL::read_off_points . . . . page 3513
CGAL::write_off_points . . . . page 3518

Example

See *read_write_xyz_point_set_example.cpp*.

Part XIII

Spatial Searching and Sorting

Chapter 59

2D Range and Neighbor Search

Matthias Bäsken

Contents

59.1 Introduction

Geometric queries are fundamental to many applications in computational geometry. The task is to maintain a dynamic set of geometric objects in such a way that certain queries can be performed efficiently. Typical examples of queries are: find out whether a given object is contained in the set, find all objects of the set lying in a given area (e.g. rectangle), find the object closest to a given point or find the pair of objects in the set lying closest to each other. Furthermore, the set should be dynamic in the sense that deletions and insertions of objects can be performed efficiently.

In computational geometry literature one can find many different data structures for maintaining sets of geometric objects. Most of them are data structures that have been developed to support a single very special kind of query operation. Examples are Voronoi diagrams for answering nearest neighbor searches, range trees for orthogonal range queries, partition trees for more general range queries, hierarchical triangulations for point location and segment trees for intersection queries ....

In many applications, different types of queries have to be performed on the same set of objects. A naive approach to this problem would use a collection of the above mentioned data structures to represent the set of objects and delegate every query operation to the corresponding structure. However, this is completely impractical since it uses too much memory and requires the maintenance of all these data structures in the presence of update operations.

Data structures that are non-optimal in theory seem to perform quite well in practice for many of these queries. For example, the Delaunay diagram turns out to be a very powerful data structure for storing dynamic sets of points under range and nearest neighbor queries. A first implementation and computational study of using Delaunay diagrams for geometric queries is described by Mehlhorn and Näher in [MN00].

In this section we present a generic variant of a two dimensional point set data type supporting various geometric queries.

The *CGAL::Point_set_2* class in this section is inherited from the two-dimensional CGAL *Delaunay Triangulation* data type.

The *CGAL::Point_set_2* class depends on two template parameters *T1* and *T2*. They are used as template parameters for the *CGAL::Delaunay_triangulation_2* class *CGAL::Point_set_2* is inherited from. *T1* is a model for the geometric traits and *T2* is a model for the triangulation data structure that the Delaunay triangulation expects.

The *CGAL::Point_set_2* class supports the following kinds of queries:

- circular range search
- triangular range search
- isorectangular range search
- (k) nearest neighbor(s)

For details about the running times see [MN00].

59.2 Example: Range Search

The following example program demonstrates the various range search operations of the two dimensional point set. First we construct a two dimensional point set *PSet* and initialize it with a few points. Then we perform circular, triangular and isorectangular range search operations on the point set.

```
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/Point_set_2.h>
#include <list>

typedef CGAL::Exact_predicates_inexact_constructions_kernel  K;

typedef CGAL::Point_set_2<K>::Vertex_handle  Vertex_handle;
typedef K::Point_2                           Point_2;

int main()
{
  CGAL::Point_set_2<K> PSet;
  std::list<Point_2> Lr;

  Point_2 p1(12,14);
  Point_2 p2(-12,14);
  Point_2 p3(2,11);
  Point_2 p4(5,6);
  Point_2 p5(6.7,3.8);
  Point_2 p6(11,20);
  Point_2 p7(-5,6);
  Point_2 p8(12,0);
  Point_2 p9(4,31);
  Point_2 p10(-10,-10);
```

```
  Lr.push_back(p1); Lr.push_back(p2); Lr.push_back(p3);
  Lr.push_back(p4); Lr.push_back(p5); Lr.push_back(p6);
  Lr.push_back(p7); Lr.push_back(p8); Lr.push_back(p9);
  Lr.push_back(p10);

  PSet.insert(Lr.begin(),Lr.end());

  std::cout << "circular range search !\n";
  CGAL::Circle_2<K> rc(p5,p6);

  std::list<Vertex_handle> LV;
  PSet.range_search(rc, std::back_inserter(LV));

  std::list<Vertex_handle>::const_iterator it;
  for (it=LV.begin();it != LV.end(); it++)
     std::cout << (*it)->point() << "\n";

  std::cout << "triangular range search !\n";

  LV.clear();
  PSet.range_search(p1,p2,p3, std::back_inserter(LV));
  for (it=LV.begin();it != LV.end(); it++)
     std::cout << (*it)->point() << "\n";
  LV.clear();

  std::cout << "isorectangular range search !\n";
  Point_2 pt1=p10;
  Point_2 pt3=p3;
  Point_2 pt2 = Point_2(pt3.x(),pt1.y());
  Point_2 pt4 = Point_2(pt1.x(),pt3.y());

  PSet.range_search(pt1,pt2,pt3,pt4, std::back_inserter(LV));
  for (it=LV.begin();it != LV.end(); it++)
    std::cout << (*it)->point() << "\n";
  return 0;
}
```

File: examples/Point_set_2/range_search.cpp

2D Range and Neighbor Search Reference Manual

Matthias Bäsken

The two dimensional point set is a class for geometric queries. It supports circular, triangular and iso rectangular range searches and nearest neighbor searches. The point set is inherited from the CGAL Delaunay triangulation data type. That means that it is a dynamic data structure supporting the insertion and deletion of points.

This package also provides function template versions of the range search and nearest neighbor query operations. They all have to be templated by the type of a CGAL Delaunay triangulation and provide functionality similar to the corresponding member functions of the point set class.

59.3 Classified Reference Pages

Concepts

PointSetTraits

Classes

CGAL::Point_set_2

Functions

CGAL::nearest_neighbor
CGAL::nearest_neighbors
CGAL::range_search

59.4 Alphabetical List of Reference Pages

CGAL::Point_set_2<Gt,Tds>

Definition

#include <CGAL/Point_set_2.h>

An instance *PS* of the data type *Point_set_2<Gt,Tds>* is a *Delaunay Triangulation* of its vertex set. The class *Point_set_2<Gt,Tds>* is inherited from the CGAL Delaunay triangulation, and provides additional nearest neighbor query operations and range searching operations.

The *Point_set_2<Gt,Tds>* class of CGAL depends on template parameters standing for the geometric traits classes used by the point set and by the Delaunay triangulation (Gt) and for the triangulation data structure (Tds).

Types

typedef Gt::Point_2	*Point;*	the point type
typedef Gt::Segment_2	*Segment;*	the segment type
typedef Gt::Circle_2	*Circle;*	the circle type
typedef Gt::FT	*Numb_type;*	the representation field number type.
Point_set_2<Gt,Tds>:: Triangulation		the underlying triangulation type.

Point_set_2<Gt,Tds>:: Triangulation::size_type

the size type of the underlying triangulation.

Point_set_2<Gt,Tds>:: Triangulation::Vertex

the vertex type of the underlying triangulation.

Point_set_2<Gt,Tds>:: Triangulation::Edge

the edge type of the underlying triangulation.

Point_set_2<Gt,Tds>:: Triangulation::Vertex_handle

handles to vertices.

Creation

Point_set_2<Gt,Tds> PS; creates an empty *Point_set_2<Gt,Tds>* .

template<class InputIterator>
Point_set_2<Gt,Tds> PS(InputIterator first, InputIterator last);

creates a *Point_set_2<Gt,Tds> PS* of the points in the range $[first, last)$.

Operations

Vertex_handle *PS.lookup(Point p)*

if *PS* contains a Vertex v with $|pos(v)| = p$ the result is a handle to v otherwise the result is $NULL$.

Vertex_handle *PS.nearest_neighbor(Point p)*

computes a handle to a vertex v of *PS* that is closest to p. If *PS* is empty, $NULL$ is returned.

Vertex_handle *PS.nearest_neighbor(Vertex_handle v)*

computes a handle to a vertex w of *PS* that is closest to v. If v is the only vertex in *PS* , $NULL$ is returned.

template<class OutputIterator>
OutputIterator *PS.nearest_neighbors(Point p, size_type k, OutputIterator res)*

computes the k nearest neighbors of p in *PS*, and places the handles to the corresponding vertices as a sequence of objects of type Vertex_handle in a container of value type of *res* which points to the first object in the sequence. The function returns an output iterator pointing to the position beyond the end of the sequence.

template<class OutputIterator>
OutputIterator *PS.nearest_neighbors(Vertex_handle v, size_type k, OutputIterator res)*

computes the k nearest neighbors of v, and places them as a sequence of objects of type Vertex_handle in a container of value type of *res* which points to the first object in the sequence. The function returns an output iterator pointing to the position beyond the end of the sequence.

template<class OutputIterator>

OutputIterator *PS.range_search(Circle C, OutputIterator res)*

computes handles to all vertices contained in the closure of disk *C*. The computed vertex handles will be placed as a sequence of objects in a container of value type of *res* which points to the first object in the sequence. The function returns an output iterator pointing to the position beyond the end of the sequence.

template<class OutputIterator>
OutputIterator *PS.range_search(Point a, Point b, Point c, OutputIterator res)*

computes handles to all vertices contained in the closure of the triangle (a,b,c).

Precondition: a, b, and c must not be collinear. The computed vertex handles will be placed as a sequence of objects in a container of value type of *res* which points to the first object in the sequence. The function returns an output iterator pointing to the position beyond the end of the sequence.

template<class OutputIterator>
OutputIterator *PS.range_search(Point a1, Point b1, Point c1, Point d1, OutputIterator res)*

computes handles to all vertices contained in the closure of the iso-rectangle $(a1,b1,c1,d1)$.

Precondition: $a1$ is the upper left point, $b1$ the lower left, $c1$ the lower right and $d1$ the upper right point of the iso rectangle. The computed vertex handles will be placed as a sequence of objects in a container of value type of *res* which points to the first object in the sequence. The function returns an output iterator pointing to the position beyond the end of the sequence.

Concept

PointSetTraits

A point set traits class has to provide some primitives that are used by the point set class. The following catalog lists the involved primitives. For details about these types see the Kernel traits documentation.

Types

PointSetTraits:: Point_2

PointSetTraits:: Circle_2

PointSetTraits:: Segment_2

PointSetTraits:: FT

PointSetTraits:: Orientation_2

PointSetTraits:: Side_of_oriented_circle_2

PointSetTraits:: Construct_circle_2

PointSetTraits:: Compute_squared_distance_2

PointSetTraits:: Bounded_side_2

PointSetTraits:: Compare_distance_2

PointSetTraits:: Construct_center_2

CGAL::nearest_neighbor

Function

Definition

The function *nearest_neighbor* is the function template version of the nearest neighbor search on Delaunay triangulations.

#include <CGAL/nearest_neighbor_delaunay_2.h>

template<class Dt>
Dt::Vertex_handle *nearest_neighbor(Dt delau, Dt::Vertex_handle v)*

computes a handle to a vertex w of $delau$ that is closest to v. If v is the only vertex in $delau$, $NULL$ is returned.

Requirements

Dt is a CGAL Delaunay triangulation and contains the following subset of types from the concept *PointSetTraits* and from the Delaunay triangulation data type:

- *Dt::Geom_traits*
- *Dt::Point*
- *Dt::Vertex_circulator*
- *Dt::Vertex_handle*
- *Dt::Geom_traits::Compare_distance_2*

Function

CGAL::nearest_neighbors

Definition

The function *nearest_neighbors* is the function template version of the k nearest neighbors search on Delaunay triangulations. There are two versions of this function, one taking a point of the Delaunay triangulation and the other taking a vertex handle.

#include <CGAL/nearest_neighbor_delaunay_2.h>

template<class Dt, class OutputIterator>
OutputIterator *nearest_neighbors(Dt& delau, Dt::Point p, Dt::size_type k, OutputIterator res)*

computes the k nearest neighbors of p in *delau*, and places the handles to the corresponding vertices as a sequence of objects of type Vertex_handle in a container of value type of *res* which points to the first object in the sequence. The function returns an output iterator pointing to the position beyond the end of the sequence.

Requirements

Dt is a CGAL Delaunay triangulation and contains the following subset of types from the concept *PointSetTraits* and from the Delaunay triangulation data type:

- *Dt::Geom_traits*
- *Dt::Vertex_handle*
- *Dt::Vertex_iterator*
- *Dt::Vertex_circulator*
- *Dt::Vertex*
- *Dt::Face*
- *Dt::Face_handle*
- *Dt::Locate_type*
- *Dt::Point*
- *Dt::Geom_traits::FT*
- *Dt::Geom_traits::Compute_squared_distance_2*

template<class Dt, class OutputIterator>

OutputIterator	*nearest_neighbors(Dt& delau, Dt::Vertex_handle v, Dt::size_type k, OutputIterator res)*

computes the k nearest neighbors of v (including v) in $delau$, and places them as a sequence of objects of type Vertex_handle in a container of value type of *res* which points to the first object in the sequence. The function returns an output iterator pointing to the position beyond the end of the sequence.

Requirements

Dt is a CGAL Delaunay triangulation and contains the following subset of types from the concept *PointSetTraits* and from the Delaunay triangulation data type:

- *Dt::Geom_traits*
- *Dt::Vertex_handle*
- *Dt::Vertex_iterator*
- *Dt::Vertex_circulator*
- *Dt::Vertex*
- *Dt::Point*
- *Dt::Geom_traits::FT*
- *Dt::Geom_traits::Compute_squared_distance_2*

Function

CGAL::range_search

Definition

There are six versions of the function template *range_search* that perform range searches on Delaunay triangulations. The first performs circular range searches, the second triangular range searches and the third performs iso-rectangular range searches. The other three range search function templates perform enhanced variants of the three aforementioned operations.
They get a user-defined object that has to control the range search operation. This way one can for instance stop the search, when n points were found.

#include <CGAL/range_search_delaunay_2.h>

template<class Dt, class Circle, class OutputIterator>
OutputIterator *range_search(Dt& delau, Circle C, OutputIterator res)*

computes handles to all vertices contained in the closure of disk C. The computed vertex handles will be placed as a sequence of objects in a container of value type of *res* which points to the first object in the sequence. The function returns an output iterator pointing to the position beyond the end of the sequence. *delau* is the CGAL Delaunay triangulation on which we perform the range search operation.

Requirements

- *Dt* is a CGAL Delaunay triangulation and contains the following subset of types from the concept *PointSetTraits* and from the Delaunay triangulation data type:
  - *Dt::Geom_traits*
  - *Dt::Vertex_handle*
  - *Dt::Vertex*
  - *Dt::Vertex_circulator*
  - *Dt::Vertex_iterator*
  - *Dt::Point*
  - *Dt::Geom_traits::Bounded_side_2*
  - *Dt::Geom_traits::Construct_center_2*
- the template parameter *Circle* corresponds to *Dt::Geom_traits::Cricle_2*

template<class Dt, class OutputIterator>

OutputIterator *range_search(Dt& delau, Dt::Point a, Dt::Point b, Dt::Point c, OutputIterator res)*

computes handles to all vertices contained in the closure of the triangle (a,b,c).

Precondition: *a*, *b*, and *c* must not be collinear. The computed vertex handles will be placed as a sequence of objects in a container of value type of *res* which points to the first object in the sequence. The function returns an output iterator pointing to the position beyond the end of the sequence. *delau* is the CGAL Delaunay triangulation on which we perform the range search operation.

Requirements

Dt is a CGAL Delaunay triangulation and contains the following subset of types from the concept *PointSetTraits* and from the Delaunay triangulation data type:

- *Dt::Geom_traits*
- *Dt::Vertex_handle*
- *Dt::Vertex*
- *Dt::Vertex_circulator*
- *Dt::Vertex_iterator*
- *Dt::Point*
- *Dt::Geom_traits::Circle_2*
- *Dt::Geom_traits::Bounded_side_2*
- *Dt::Geom_traits::Construct_center_2*
- *Dt::Geom_traits::Orientation_2*
- *Dt::Geom_traits::Construct_circle_2*

template<class Dt, class OutputIterator>
OutputIterator *range_search(Dt& delau,*
Dt::Point a,
Dt::Point b,
Dt::Point c,
Dt::Point d,

OutputIterator res)

computes handles to all vertices contained in the closure of the iso-rectangle (a,b,c,d).

Precondition: a is the upper left point, b the lower left, c the lower right and d the upper right point of the iso rectangle. The computed vertex handles will be placed as a sequence of objects in a container of value type of *res* which points to the first object in the sequence. The function returns an output iterator pointing to the position beyond the end of the sequence. $delau$ is the CGAL Delaunay triangulation on which we perform the range search operation.

Requirements

Dt is a CGAL Delaunay triangulation and contains the following subset of types from the concept *PointSetTraits* and from the Delaunay triangulation data type:

- *Dt::Geom_traits*
- *Dt::Vertex_handle*
- *Dt::Vertex*
- *Dt::Vertex_circulator*
- *Dt::Vertex_iterator*
- *Dt::Point*
- *Dt::Geom_traits::Circle_2*
- *Dt::Geom_traits::Bounded_side_2*
- *Dt::Geom_traits::Construct_center_2*
- *Dt::Geom_traits::Orientation_2*
- *Dt::Geom_traits::Construct_circle_2*

template<class Dt, class Circle, class OutputIterator, class Pred>
OutputIterator *range_search(Dt& delau,*
Circle C,
OutputIterator res,
Pred& pred,
bool return_if_succeded)

computes handles to all vertices contained in the closure of disk C. The computed vertex handles will be placed as a sequence of objects in a container of value type of *res* which points to the first object in the sequence. The function returns an output iterator pointing to the position beyond the end of the sequence. $delau$ is the CGAL Delaunay triangulation on that we perform the range search operation. *pred* controls the search operation. If $return\_if\_succeded$ is $true$, we will end the search after the first success of the predicate, otherwise we will continue till the search is finished.

Requirements

For the requirements of *Dt* see the description for the non-predicate version.
Requirements of *Pred*:

- *void set_result(bool);*
- *bool operator()(const Point&);*

The *operator()* is used for testing the current point in the search operation. If this operator returns *true* and *return_if_succeded* is *true*, the range search will stop. Otherwise the range search operation will continue. Member function *set_result* can be used to store the result of the range search in the function object. The result will be *true* if the last call to the *operator()* of the predicate returned *true*, *false* otherwise.

template<class Dt, class OutputIterator, class Pred>
OutputIterator range_search(Dt& delau, Dt::Point a, Dt::Point b, Dt::Point c, OutputIterator res, Pred& pred, bool return_if_succeded)

computes handles to all vertices contained in the closure of the triangle (a,b,c).

Precondition: *a*, *b*, and *c* must not be collinear. The computed vertex handles will be placed as a sequence of objects in a container of value type of *res* which points to the first object in the sequence. The function returns an output iterator pointing to the position beyond the end of the sequence. *delau* is the CGAL Delaunay triangulation on which we perform the range search operation. *pred* controls the search operation. If *return_if_succeded* is *true*, we will end the search after the first success of the predicate, otherwise we will continue till the search is finished.

Requirements

For the requirements of *Dt* see the description for the non-predicate version.
For the requirements of *Pred* see the description above.

template<class Dt, class OutputIterator, class Pred>
OutputIterator range_search(Dt& delau, Dt::Point a, Dt::Point b, Dt::Point c, Dt::Point d, OutputIterator res, Pred& pred,

bool return_if_succeded)

computes handles to all vertices contained in the closure of the iso-rectangle (a,b,c,d).

Precondition: *a* is the upper left point, *b* the lower left, *c* the lower right and *d* the upper right point of the iso rectangle. The computed vertex handles will be placed as a sequence of objects in a container of value type of *res* which points to the first object in the sequence. The function returns an output iterator pointing to the position beyond the end of the sequence. $delau$ is the CGAL Delaunay triangulation on which we perform the range search operation. *pred* controls the search operation. If $return\_if\_succeded$ is $true$, we will end the search after the first success of the predicate, otherwise we will continue till the search is finished.

Requirements

For the requirements of *Dt* see the description for the non-predicate version.
For the requirements of *Pred* see the description above.

Chapter 60

Interval Skip List

Andreas Fabri

Contents

60.1 Definition

An interval skip list is a data structure for finding all intervals that contain a point, and for stabbing queries, that is for answering the question whether a given point is contained in an interval or not. The implementation we provide is dynamic, that is the user can freely mix calls to the methods *insert(..)*, *remove(..)*, *find_intervals(..)*, and *is_contained(..)*

The interval skip list class is parameterized with an interval class.

The data structure was introduced by Hanson [Han91], and it is called interval skip list, because it is an extension of the randomized list structure known as skip list [Pug90].

60.2 Example Programs

We give two examples. The first one uses a basic interval class. In the second example we create an interval skip list for the z-ranges of the faces of a terrain, allowing to answer level queries.

60.2.1 First Example with Simple Interval

The first example reads two numbers n and d from *std::cin*. It creates n intervals of length d with the left endpoint at n. It then reads out the intervals for the 1-dimensional points with coordinates $0 \ldots n+d$.

The interval skip list class has as template argument an interval class. In this example we use the class *Interval_skip_list_interval*.

```
#include <CGAL/Interval_skip_list.h>
#include <CGAL/Interval_skip_list_interval.h>
#include <vector>
#include <list>
#include <iostream>

typedef CGAL::Interval_skip_list_interval<double> Interval;
typedef CGAL::Interval_skip_list<Interval> Interval_skip_list;

int main()
{
  Interval_skip_list isl;
  int i, n, d;

  n = 10;
  d = 3;
  //std::cin >> n >> d;
  std::vector<Interval> intervals(n);
  for(i = 0; i < n; i++) {
    intervals[i] = Interval(i, i+d);
  }
  std::random_shuffle(intervals.begin(), intervals.end());

  isl.insert(intervals.begin(), intervals.end());

  for(i = 0; i < n+d; i++) {
    std::list<Interval> L;
    isl.find_intervals(i, std::back_inserter(L));
    for(std::list<Interval>::iterator it = L.begin(); it != L.end();
   it++){
      std::cout << *it;
    }
    std::cout << std::endl;
  }

  for(i = 0; i < n; i++) {
    isl.remove(intervals[i]);
  }
  return 0;

}
```

File: examples/Interval_skip_list/intervals.cpp

60.2.2 Example with Faces of a Triangulated Terrain

The second example creates an interval skip list that allows to find all the faces of a terrain intersected by an horizontal plane at a given height. The data points for the terrain are read from a file.

As model for the interval concept, we use a class that is a wrapper around a face handle of a triangulated terrain. Lower and upper bound of the interval are smallest and largest z-coordinate of the face.

```
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/Delaunay_triangulation_2.h>
#include <CGAL/Projection_traits_xy_3.h>
#include <CGAL/Interval_skip_list.h>
#include <CGAL/Level_interval.h>
#include <iostream>
#include <fstream>

typedef CGAL::Exact_predicates_inexact_constructions_kernel EIK;
typedef EIK::Point_3                                    Point_3;
typedef CGAL::Projection_traits_xy_3<EIK> K;
typedef CGAL::Delaunay_triangulation_2<K>               Delaunay;
typedef Delaunay::Face_handle                           Face_handle;
typedef Delaunay::Finite_faces_iterator
   Finite_faces_iterator;
typedef CGAL::Level_interval<Face_handle>               Interval;
typedef CGAL::Interval_skip_list<Interval>
   Interval_skip_list;

int main()
{
  std::ifstream fin("terrain.pts"); // elevation ranges from 0 to 100
  Delaunay dt;

  dt.insert(std::istream_iterator<Point_3>(fin),
            std::istream_iterator<Point_3>());

  Interval_skip_list isl;
  for(Finite_faces_iterator fh = dt.finite_faces_begin();
      fh != dt.finite_faces_end();
      ++fh){
    isl.insert(Interval(fh));
  }
  std::list<Interval> level;
  isl.find_intervals(50, std::back_inserter(level));
  for(std::list<Interval>::iterator it = level.begin();
      it != level.end();
      ++it){
    std::cout << dt.triangle(it->face_handle()) << std::endl;
  }
  return 0;
}
```

File: examples/Interval_skip_list/isl_terrain.cpp

Interval Skip List Reference Manual

Andreas Fabri

This chapter presents the interval skip list introduced by Hanson [Han91], and derived from the skip list data structure [Pug90].

The data structure stores intervals and allows to perform stabbing queries, that is to test whether a point is covered by any of the intervals. It further allows to find all intervals that contain a point.

The interval skip list is, as far as its functionality is concerned, related to the *Segment_tree*. Both allow to do stabbing queries and both allow to find all intervals that contain a given point. The implementation of segment trees in CGAL works in higher dimensions, whereas the interval skip list is limited to the 1D case. However, this interval skip list implementation is fully dynamic, whereas the segment tree implementation in CGAL is static, that is all intervals must be known in advance.

This package has one concept, namely for the interval with which the interval skip list class is parameterized.

60.3 Classified Reference Pages

Concepts

Classes

60.4 Alphabetical List of Reference Pages

CGAL::Interval_skip_list<Interval>

Definition

The class *Interval_skip_list<Interval>* is a dynamic data structure that allows to find all members of a set of intervals that overlap a point.

#include <CGAL/Interval_skip_list.h>

Types

typedef Interval::Value	*Value;*	the type of inf and sup of the interval.

Interval_skip_list<Interval>:: const_iterator

An iterator over all intervals.

Creation

Interval_skip_list<Interval> isl;

Default constructor.

template < class InputIterator >
Interval_skip_list<Interval> isl(InputIterator first, InputIterator last);

Constructor that inserts the iterator range *[first, last)* in the interval skip list.
Precondition: The *value_type* of *first* and *last* is *Interval*.

Operations

template < class InputIterator >
int *isl.insert(InputIterator first, InputIterator last)*

Inserts the iterator range *[first, last)* in the interval skip list, and returns the number of inserted intervals.
Precondition: The *value_type* of *first* and *last* is *Interval*.

void *isl.insert(Interval i)*

inserts interval *i* in the interval skip list.

bool *isl.remove(Interval i)*

removes interval *i* from the interval skip list. Returns *true* iff removal was successful.

bool *isl.is_contained(Value v)*

Returns *true* iff there is an interval that contains *v*.

template < class OutputIterator >
OutputIterator *isl.find_intervals(Value v, OutputIterator out)*

Writes the intervals *i* with *i.inf()* $\leq v \leq$ *i.sup* to the output iterator *out*.
Precondition: The *value_type* of *out* is *Interval*.

void *isl.clear()*

Removes all intervals from the interval skip list.

const_iterator *isl.begin() const*

Returns an iterator over all intervals.

const_iterator *isl.end() const*

Returns the past the end iterator.

I/O

ostream& *ostream& os << isl*

Inserts the interval skip list *isl* into the stream *os*.
Precondition: The output operator must be defined for *Interval*.

Implementation

The insertion and deletion of a segment in the interval skip list takes expected time $O(\log^2 n)$, if the segment endpoints are chosen from a continuous distribution. A stabbing query takes expected time $O(\log n)$, and finding all intervals that contain a point takes expected time $O(\log n + k)$, where k is the number of intervals.

The implementation is based on the code developed by Eric N. Hansen, which can be found at `http://www-pub.cise.ufl.edu/~hanson/IS-lists/`. Attention, this code has memory leaks.

Interval

Definition

The concept Interval describes the requirements for the template argument *Interval* of a *Interval_skip_list< Interval>*.

The concept does not specify, whether the interval is open or closed. It is up to the implementer of a model for this concept to define that.

Creation

Interval in;	default constructor.

Types

Interval:: Value	The type of the lower and upper bound of the interval.

Access Functions

Value	*in.inf() const*	returns the lower bound.
Value	*in.sup() const*	returns the upper bound.
bool	*in.contains(Value v) const*	
		returns *true*, iff *in* contains *v*.
bool	*in.contains_interval(Value i, Value s) const*	
		returns *true*, iff *in* contains *(i,s)*.
bool	$in == I$	Equality test.
bool	$in \,!= I$	Unequality test.

Has Models

CGAL::Interval_skip_list_interval<Value>
CGAL::Face_interval

See Also

Interval_skip_list

Class

CGAL::Interval_skip_list_interval<Value>

Definition

The class *Interval_skip_list_interval<Value>* represents intervals with lower and upper bound of type *Value*. These intervals can be opened or closed at each endpoint.

#include <CGAL/Interval_skip_list_interval.h>

Creation

Interval_skip_list_interval<Value> i;

Default constructor.

Interval_skip_list_interval<Value> i(Value i, Value s, bool ic = true, bool uc = true);

Constructs the interval with infimum *i* and supremum *s*. The arguments *ic* and *uc* have value *true*, iff the interval is closed at the lower and upper bound, respectively.

Operations

bool *i.inf_closed() const*

returns *true*, iff the interval is closed at the lower bound.

bool *i.sup_closed() const*

returns *true*, iff the interval is closed at the upper bound.

I/O

The output operator is defined for *std::ostream*.

ostream& *ostream& os << Interval_skip_list_interval<V> i*

Inserts the interval *i* into the stream *os*.

Precondition: The output operator for *Value* is defined.

Is Model for the Concepts

Interval

CGAL::Level_interval<FaceHandle>

Class

Definition

The class *Level_interval<FaceHandle>* represents intervals for the minimum and maximum value of the z-coordinate of a face of a triangulation.

#include <CGAL/Level_interval.h>

Requirements

The *value_type* of *FaceHandle* must be *Face*, which must have a nested type *Vertex*, which must have a nested type *Point*, whose *Kernel_traits<Point>Kernel* must have a nested type *FT*. These requirements are fulfilled, if one uses a CGAL triangulation and a CGAL kernel.

Types

typedef FT	*Value;*	The type of the z-coordinate of points stored in vertices of faces.

Creation

Level_interval<FaceHandle> i; — Default constructor.

Level_interval<FaceHandle> i(FaceHandle fh);

Constructs the interval with smallest and largest z coordinate of the points stored in the vertices of the face *fh* points to.

Operations

FaceHandle	*i.face_handle()*	returns the face handle.

I/O

ostream&	*ostream& os << i*	Inserts the interval *i* into the stream *os*. *Precondition*: The output operator for **Face_handle* is defined.

Is Model for the Concepts

Interval

Chapter 61

dD Spatial Searching

Hans Tangelder and Andreas Fabri

Contents

61.1 Introduction

The spatial searching package implements exact and approximate distance browsing by providing implementations of algorithms supporting

- both nearest and furthest neighbor searching
- both exact and approximate searching
- (approximate) range searching
- (approximate) k-nearest and k-furthest neighbor searching

- (approximate) incremental nearest and incremental furthest neighbor searching
- query items representing points and spatial objects.

In these searching problems a set P of data points in d-dimensional space is given. The points can be represented by Cartesian coordinates or homogeneous coordinates. These points are preprocessed into a tree data structure, so that given any query item q the points of P can be browsed efficiently. The approximate spatial searching package is designed for data sets that are small enough to store the search structure in main memory (in contrast to approaches from databases that assume that the data reside in secondary storage).

61.1.1 Neighbor Searching

Spatial searching supports browsing through a collection of d-dimensional spatial objects stored in a spatial data structure on the basis of their distances to a query object. The query object may be a point or an arbitrary spatial object, e.g, a d-dimensional sphere. The objects in the spatial data structure are d-dimensional points.

Often the number of the neighbors to be computed is not know beforehand, e.g., because the number may depend on some properties of the neighbors (for example when querying for the nearest city to Paris with population greater than a million) or the distance to the query point. The conventional approach is *k-nearest neighbor searching* that makes use of a k-nearest neighbor algorithm, where k is known prior to the invocation of the algorithm. Hence, the number of nearest neighbors has to be guessed. If the guess is too large redundant computations are performed. If the number is too small the computation has to be re-invoked for a larger number of neighbors, thereby performing redundant computations. Therefore, Hjaltason and Samet [HS95] introduced *incremental nearest neighbor searching* in the sense that having obtained the k nearest neighbors, the $k + 1^{st}$ neighbor can be obtained without having to calculate the $k + 1$ nearest neighbor from scratch.

Spatial searching typically consists of a preprocessing phase and a searching phase. In the preprocessing phase one builds a search structure and in the searching phase one makes the queries. In the preprocessing phase the user builds a tree data structure storing the spatial data. In the searching phase the user invokes a searching method to browse the spatial data.

With relatively minor modifications, nearest neighbor searching algorithms can be used to find the furthest object from the query object. Therefore, *furthest neighbor searching* is also supported by the spatial searching package.

The execution time for exact neighbor searching can be reduced by relaxing the requirement that the neighbors should be computed exactly. If the distances of two objects to the query object are approximately the same, instead of computing the nearest/furthest neighbor exactly, one of these objects may be returned as the approximate nearest/furthest neighbor. I.e., given some non-negative constant ε the distance of an object returned as an approximate k-nearest neighbor must not be larger than $(1+\varepsilon)r$, where r denotes the distance to the real k^{th} nearest neighbor. Similar the distance of an approximate k-furthest neighbor must not be smaller than $r/(1+\varepsilon)$. Obviously, for $\varepsilon = 0$ we get the exact result, and the larger ε is, the less exact the result.

Neighbor searching is implemented by the following four classes.

The class *CGAL::Orthogonal_k_neighbor_search<Traits, OrthogonalDistance, Splitter, SpatialTree>* implements the standard search strategy for orthogonal distances like the weighted Minkowski distance. It requires the use of extended nodes in the spatial tree and supports only k neighbor searching for point queries.

The class *CGAL::K_neighbor_search<Traits, GeneralDistance, Splitter, SpatialTree>* implements the standard search strategy for general distances like the Manhattan distance for iso-rectangles. It does not require the use of extended nodes in the spatial tree and supports only k neighbor searching for queries defined by points or spatial objects.

The class *Orthogonal_incremental_neighbor_search<Traits, GeneralDistance, Splitter, SpatialTree>* implements the incremental search strategy for general distances like the weighted Minkowski distance. It requires the use of extended nodes in the spatial tree and supports incremental neighbor searching and distance browsing for point queries.

The class *CGAL::Incremental_neighbor_search<Traits, GeneralDistance, Splitter, SpatialTree>* implements the incremental search strategy for general distances like the Manhattan distance for iso-rectangles. It does not require the use of extended nodes in the spatial tree and supports incremental neighbor searching and distance browsing for queries defined by points or spatial objects.

61.1.2 Range Searching

Exact range searching and *approximate range searching* is supported using exact or fuzzy d-dimensional objects enclosing a region. The fuzziness of the query object is specified by a parameter ε denoting a maximal allowed distance to the boundary of a query object. If the distance to the boundary is at least ε, points inside the object are always reported and points outside the object are never reported. Points within distance ε to the boundary may be or may be not reported. For exact range searching the fuzziness parameter ε is set to zero.

The class *Kd_tree* implements range searching in the method *search*, which is a template method with an output iterator and a model of the concept *FuzzyQueryItem* as *CGAL::Fuzzy_iso_box_d* or *CGAL::Fuzzy_sphere_d*. For range searching of large data sets the user may set the parameter *bucket_size* used in building the *k-d* tree to a large value (e.g. 100), because in general the query time will be less then using the default value.

61.2 Splitting Rules

Instead of using the default splitting rule *Sliding_midpoint* described below, a user may, depending upon the data, select one from the following splitting rules, which determine how a separating hyperplane is computed:

Midpoint_of_rectangle This splitting rule cuts a rectangle through its midpoint orthogonal to the longest side.

Midpoint_of_max_spread This splitting rule cuts a rectangle through $(Mind + Maxd)/2$ orthogonal to the dimension with the maximum point spread $[Mind, Maxd]$.

Sliding_midpoint This is a modification of the midpoint of rectangle splitting rule. It first attempts to perform a midpoint of rectangle split as described above. If data points lie on both sides of the separating plane the sliding midpoint rule computes the same separator as the midpoint of rectangle rule. If the data points lie only on one side it avoids this by sliding the separator, computed by the midpoint of rectangle rule, to the nearest data point.

Median_of_rectangle The splitting dimension is the dimension of the longest side of the rectangle. The splitting value is defined by the median of the coordinates of the data points along this dimension.

Median_of_max_spread The splitting dimension is the dimension of the longest side of the rectangle. The splitting value is defined by the median of the coordinates of the data points along this dimension.

Fair This splitting rule is a compromise between the median of rectangle splitting rule and the midpoint of rectangle splitting rule. This splitting rule maintains an upper bound on the maximal allowed ratio of the longest and shortest side of a rectangle (the value of this upper bound is set in the constructor of the fair splitting rule). Among the splits that satisfy this bound, it selects the one in which the points have the largest spread. It then splits the points in the most even manner possible, subject to maintaining the bound on the ratio of the resulting rectangles.

Sliding_fair This splitting rule is a compromise between the fair splitting rule and the sliding midpoint rule. Sliding fair-split is based on the theory that there are two types of splits that are good: balanced splits that produce fat rectangles, and unbalanced splits provided the rectangle with fewer points is fat.

Also, this splitting rule maintains an upper bound on the maximal allowed ratio of the longest and shortest side of a rectangle (the value of this upper bound is set in the constructor of the fair splitting rule). Among the splits that satisfy this bound, it selects the one one in which the points have the largest spread. It then considers the most extreme cuts that would be allowed by the aspect ratio bound. This is done by dividing the longest side of the rectangle by the aspect ratio bound. If the median cut lies between these extreme cuts, then we use the median cut. If not, then consider the extreme cut that is closer to the median. If all the points lie to one side of this cut, then we slide the cut until it hits the first point. This may violate the aspect ratio bound, but will never generate empty cells.

61.3 Example Programs

We give six examples. The first example illustrates k nearest neighbor searching, and the second example incremental neighbor searching. The third is an example of approximate furthest neighbor searching using a *d*-dimensional iso-rectangle as an query object. Approximate range searching is illustrated by the fourth example. The fifth example illustrates k neighbor searching for a user defined point class. The last example shows how to choose another splitting rule in the *k*-*d* tree that is used as search tree.

61.3.1 Example for K Neighbor Searching

The first example illustrates k neighbor searching with an Euclidean distance and 2-dimensional points. The generated random data points are inserted in a search tree. We then initialize the k neighbor search object with the origin as query. Finally, we obtain the result of the computation in the form of an iterator range. The value of the iterator is a pair of a point and its square distance to the query point. We use square distances, or *transformed distances* for other distance classes, as they are computationally cheaper.

```
#include <CGAL/Simple_cartesian.h>
#include <CGAL/point_generators_2.h>
#include <CGAL/Orthogonal_k_neighbor_search.h>
#include <CGAL/Search_traits_2.h>
#include <list>
#include <cmath>


typedef CGAL::Simple_cartesian<double> K;
typedef K::Point_2 Point_d;
typedef CGAL::Search_traits_2<K> TreeTraits;
typedef CGAL::Orthogonal_k_neighbor_search<TreeTraits> Neighbor_search;
typedef Neighbor_search::Tree Tree;

int main() {
  const unsigned int N = 1;

  std::list<Point_d> points;
  points.push_back(Point_d(0,0));

  Tree tree(points.begin(), points.end());

  Point_d query(0,0);

  // Initialize the search structure, and search all N points

  Neighbor_search search(tree, query, N);

   // report the N nearest neighbors and their distance
  // This should sort all N points by increasing distance from origin
  for(Neighbor_search::iterator it = search.begin(); it != search.end();
   ++it){
    std::cout << it->first << " "<< std::sqrt(it->second) << std::endl;
  }


  return 0;
}
```

File: examples/Spatial_searching/nearest_neighbor_searching.cpp

61.3.2 Example for Incremental Searching

This example program illustrates incremental searching for the closest point with a positive first coordinate. We can use the orthogonal incremental neighbor search class, as the query is also a point and as the distance is the Euclidean distance.

As for the *k* neighbor search, we first initialize the search tree with the data. We then create the search object, and finally obtain the iterator with the *begin()* method. Note that the iterator is of the input iterator category, that is one can make only one pass over the data.

```
#include <CGAL/Simple_cartesian.h>
#include <CGAL/Orthogonal_incremental_neighbor_search.h>
#include <CGAL/Search_traits_2.h>

typedef CGAL::Simple_cartesian<double> K;
typedef K::Point_2 Point_d;
typedef CGAL::Search_traits_2<K> TreeTraits;
typedef CGAL::Orthogonal_incremental_neighbor_search<TreeTraits>
    NN_incremental_search;
typedef NN_incremental_search::iterator NN_iterator;
typedef NN_incremental_search::Tree Tree;

// A functor that returns true, iff the x-coordinate of a dD point is not
     positive
struct X_not_positive {
  bool operator()(const NN_iterator& it) { return ((*it).first)[0]<0;  }
};

// An iterator that only enumerates dD points with positive x-coordinate
typedef CGAL::Filter_iterator<NN_iterator, X_not_positive>
    NN_positive_x_iterator;

int main() {

  Tree tree;
  tree.insert(Point_d(0,0));
  tree.insert(Point_d(1,1));
  tree.insert(Point_d(0,1));
  tree.insert(Point_d(10,110));
  tree.insert(Point_d(45,0));
  tree.insert(Point_d(0,2340));
  tree.insert(Point_d(0,30));

  Point_d query(0,0);

  NN_incremental_search NN(tree, query);
  NN_positive_x_iterator it(NN.end(), X_not_positive(), NN.begin()),
    end(NN.end(), X_not_positive());

  std::cout <<  "The first 5 nearest neighbours with positive x-coord
    are: " << std::endl;
  for (int j=0; (j < 5)&&(it!=end); ++j,++it)
    std::cout <<   (*it).first << "  at squared distance = " <<
   (*it).second << std::endl;
```

```
  return 0;
}
```

File: examples/Spatial_searching/distance_browsing.cpp

61.3.3 Example for General Neighbor Searching

This example program illustrates approximate nearest and furthest neighbor searching using 4-dimensional Cartesian coordinates. Five approximate nearest neighbors of the query rectangle $[0.1, 0.2]^4$ are computed. Because the query object is a rectangle we cannot use the Orthogonal neighbor search. As in the previous examples we first initialize a search tree, create the search object with the query, and obtain the result of the search as iterator range.

```
#include <CGAL/Cartesian_d.h>
#include <CGAL/point_generators_2.h>
#include <CGAL/Manhattan_distance_iso_box_point.h>
#include <CGAL/K_neighbor_search.h>
#include <CGAL/Search_traits_2.h>

typedef CGAL::Cartesian_d<double> K;
typedef K::Point_d Point_d;
typedef CGAL::Random_points_in_square_2<Point_d> Random_points_iterator;
typedef K::Iso_box_d Iso_box_d;
typedef K TreeTraits;
typedef CGAL::Manhattan_distance_iso_box_point<TreeTraits> Distance;
typedef CGAL::K_neighbor_search<TreeTraits, Distance> Neighbor_search;
typedef Neighbor_search::Tree Tree;

int  main() {
  const int N = 1000;
  const unsigned int K = 10;

  Tree tree;
  Random_points_iterator rpg;
  for(int i = 0; i < N; i++){
    tree.insert(*rpg++);
  }
  Point_d pp(0.1,0.1);
  Point_d qq(0.2,0.2);
  Iso_box_d query(pp,qq);

  Distance tr_dist;
  Neighbor_search N1(tree, query, K, 0.0, false); // eps=10.0,
   nearest=false

  std::cout << "For query rectange = [0.1,0.2]^2 " << std::endl
            <<  "The " << K << " approximate furthest neighbors are: " <<
     std::endl;
  for (Neighbor_search::iterator it = N1.begin();it != N1.end();it++) {
    std::cout << " Point " << it->first << " at distance = " <<
   tr_dist.inverse_of_transformed_distance(it->second) << std::endl;
  }
  return 0;
}
```

File: examples/Spatial_searching/general_neighbor_searching.cpp

61.3.4 Example for a Range Query

This example program illustrates approximate range querying for 4-dimensional fuzzy iso-rectangles and spheres using homogeneous coordinates. The range queries are member functions of the *k-d* tree class.

```
#include <CGAL/Cartesian_d.h>
#include <CGAL/point_generators_d.h>
#include <CGAL/Kd_tree.h>
#include <CGAL/Fuzzy_sphere.h>
#include <CGAL/Fuzzy_iso_box.h>
#include <CGAL/Search_traits_d.h>

typedef CGAL::Cartesian_d<double> K;
typedef K::Point_d Point_d;
typedef CGAL::Search_traits_d<K> Traits;
typedef CGAL::Random_points_in_cube_d<Point_d>
    Random_points_iterator;
typedef CGAL::Counting_iterator<Random_points_iterator>
    N_Random_points_iterator;
typedef CGAL::Kd_tree<Traits> Tree;
typedef CGAL::Fuzzy_sphere<Traits> Fuzzy_sphere;
typedef CGAL::Fuzzy_iso_box<Traits> Fuzzy_iso_box;

int main() {
  const int D = 4;
  const int N = 1000;
  // generator for random data points in the square ( (-1000,-1000),
    (1000,1000) )
  Random_points_iterator rpit(4, 1000.0);

  // Insert N points in the tree
  Tree tree(N_Random_points_iterator(rpit,0),
            N_Random_points_iterator(rpit,N));

  // define range query objects
  double  pcoord[D] = { 300, 300, 300, 300 };
  double  qcoord[D] = { 900.0, 900.0, 900.0, 900.0 };
  Point_d p(D, pcoord, pcoord+D);
  Point_d q(D, qcoord, qcoord+D);
  Fuzzy_sphere fs(p, 700.0, 100.0);
  Fuzzy_iso_box fib(p, q, 100.0);

  std::cout << "points approximately in fuzzy range query" << std::endl;
  std::cout << "with center (300.0, 300.0, 300.0, 300.0)" << std::endl;
  std::cout << "and fuzzy radius <200.0,400.0> are:" << std::endl;
  tree.search(std::ostream_iterator<Point_d>(std::cout, "\n"), fs);

  std::cout << "points approximately in fuzzy range query ";
  std::cout << "[<200,4000>,<800,1000>]]^4 are:" << std::endl;

  tree.search(std::ostream_iterator<Point_d>(std::cout, "\n"), fib);
  return 0;
}
```

File: examples/Spatial_searching/fuzzy_range_query.cpp

61.3.5 Example for User Defined Point and Distance Class

The neighbor searching works with all CGAL kernels, as well as with user defined points and distance classes. In this example we assume that the user provides the following 3-dimensional points class.

```
struct Point {
  double vec[3];

  Point() { vec[0]= vec[1] = vec[2] = 0; }
  Point (double x, double y, double z) { vec[0]=x; vec[1]=y; vec[2]=z;  }

  double x() const { return vec[ 0 ]; }
  double y() const { return vec[ 1 ]; }
  double z() const { return vec[ 2 ]; }

  double& x() { return vec[ 0 ]; }
  double& y() { return vec[ 1 ]; }
  double& z() { return vec[ 2 ]; }

  bool operator==(const Point& p) const
  {
    return (x() == p.x()) && (y() == p.y()) && (z() == p.z())  ;
  }

  bool  operator!=(const Point& p) const { return ! (*this == p); }
}; //end of class



namespace CGAL {

  template <>
  struct Kernel_traits<Point> {
    struct Kernel {
      typedef double FT;
      typedef double RT;
    };
  };
}


struct Construct_coord_iterator {
  typedef  const double* result_type;
  const double* operator()(const Point& p) const
  { return static_cast<const double*>(p.vec); }

  const double* operator()(const Point& p, int)  const
  { return static_cast<const double*>(p.vec+3); }
};
```

File: examples/Spatial_searching/Point.h

We have put the glue layer in this file as well, that is a class that allows to iterate over the Cartesian coordinates of the point, and a class to construct such an iterator for a point. We next need a distance class

```
struct Distance {
  typedef Point Query_item;
  typedef double FT;

  double transformed_distance(const Point& p1, const Point& p2) const {
    double distx= p1.x()-p2.x();
    double disty= p1.y()-p2.y();
    double distz= p1.z()-p2.z();
    return distx*distx+disty*disty+distz*distz;
  }

  template <class TreeTraits>
  double min_distance_to_rectangle(const Point& p,
                                   const
   CGAL::Kd_tree_rectangle<TreeTraits>& b) const {
    double distance(0.0), h = p.x();
    if (h < b.min_coord(0)) distance +=
   (b.min_coord(0)-h)*(b.min_coord(0)-h);
    if (h > b.max_coord(0)) distance +=
   (h-b.max_coord(0))*(h-b.max_coord(0));
    h=p.y();
    if (h < b.min_coord(1)) distance +=
   (b.min_coord(1)-h)*(b.min_coord(1)-h);
    if (h > b.max_coord(1)) distance +=
   (h-b.max_coord(1))*(h-b.min_coord(1));
    h=p.z();
    if (h < b.min_coord(2)) distance +=
   (b.min_coord(2)-h)*(b.min_coord(2)-h);
    if (h > b.max_coord(2)) distance +=
   (h-b.max_coord(2))*(h-b.max_coord(2));
    return distance;
  }

  template <class TreeTraits>
  double max_distance_to_rectangle(const Point& p,
                                   const
   CGAL::Kd_tree_rectangle<TreeTraits>& b) const {
    double h = p.x();

    double d0 = (h >= (b.min_coord(0)+b.max_coord(0))/2.0) ?
                (h-b.min_coord(0))*(h-b.min_coord(0)) :
   (b.max_coord(0)-h)*(b.max_coord(0)-h);

    h=p.y();
    double d1 = (h >= (b.min_coord(1)+b.max_coord(1))/2.0) ?
                (h-b.min_coord(1))*(h-b.min_coord(1)) :
   (b.max_coord(1)-h)*(b.max_coord(1)-h);
    h=p.z();
    double d2 = (h >= (b.min_coord(2)+b.max_coord(2))/2.0) ?
                (h-b.min_coord(2))*(h-b.min_coord(2)) :
   (b.max_coord(2)-h)*(b.max_coord(2)-h);
    return d0 + d1 + d2;
  }
```

```
  double new_distance(double& dist, double old_off, double new_off,
                      int /* cutting_dimension */)  const {
    return dist + new_off*new_off - old_off*old_off;
  }

  double transformed_distance(double d) const { return d*d; }

  double inverse_of_transformed_distance(double d) { return std::sqrt(d);
    }

}; // end of struct Distance
```

File: examples/Spatial_searching/Distance.h

We are ready to put the pieces together. The class *Search_traits*<..> which you see in the next file is then a mere wrapper for all these types. The searching itself works exactly as for CGAL kernels.

```
#include <CGAL/basic.h>
#include <CGAL/Search_traits.h>
#include <CGAL/point_generators_3.h>
#include <CGAL/Orthogonal_k_neighbor_search.h>
#include "Point.h"  // defines types Point, Construct_coord_iterator
#include "Distance.h"

typedef CGAL::Random_points_in_cube_3<Point> Random_points_iterator;
typedef CGAL::Counting_iterator<Random_points_iterator>
    N_Random_points_iterator;
typedef CGAL::Search_traits<double, Point, const double*,
    Construct_coord_iterator> Traits;
typedef CGAL::Orthogonal_k_neighbor_search<Traits, Distance>
    K_neighbor_search;
typedef K_neighbor_search::Tree Tree;

int main() {
  const int N = 1000;
  const unsigned int K = 5;
  // generator for random data points in the cube ( (-1,-1,-1), (1,1,1) )
  Random_points_iterator rpit( 1.0);

  // Insert number_of_data_points in the tree
  Tree tree(N_Random_points_iterator(rpit,0),
            N_Random_points_iterator(N));

  Point query(0.0, 0.0, 0.0);
  Distance tr_dist;

  // search K nearest neighbours
  K_neighbor_search search(tree, query, K);
  for(K_neighbor_search::iterator it = search.begin(); it !=
    search.end(); it++){
     std::cout << " d(q, nearest neighbor)=  "
               << tr_dist.inverse_of_transformed_distance(it->second) <<
    std::endl;
  }
  // search K furthest neighbour searching, with eps=0,
    search_nearest=false
  K_neighbor_search search2(tree, query, K, 0.0, false);

  for(K_neighbor_search::iterator it = search2.begin(); it !=
    search2.end(); it++){
     std::cout << " d(q, furthest neighbor)=  "
               << tr_dist.inverse_of_transformed_distance(it->second) <<
    std::endl;
  }
  return 0;
}
```

File: examples/Spatial_searching/user_defined_point_and_distance.cpp

61.3.6 Examples for Using an Arbitrary Point Type with Point Property Maps

The following three example programs illustrate how to use the classes *Search_traits_adapter<Key,PointPropertyMap,BaseTraits>* and *Distance_for_point_adapter<Key,PointPropertyMap,Base_distance>* to store in the kd-tree objects of an arbitrary key type. Points are accessed through a point property map. This is really convenient to associate information to a point or to reduce the size of the search structure.

Using a Point and an Integer as Key Type

In this example program, the search tree stores tuples of point and integer. The value type of the iterator of the neighbor searching algorithm is this tuple type.

```
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/basic.h>
#include <CGAL/Search_traits_3.h>
#include <CGAL/Search_traits_adapter.h>
#include <CGAL/point_generators_3.h>
#include <CGAL/Orthogonal_k_neighbor_search.h>
#include <boost/iterator/zip_iterator.hpp>
#include <utility>

typedef CGAL::Exact_predicates_inexact_constructions_kernel Kernel;
typedef Kernel::Point_3 Point_3;
typedef boost::tuple<Point_3,int> Point_and_int;

//definition of the property map
struct My_point_property_map{
  typedef Point_3 value_type;
  typedef const value_type& reference;
  typedef const Point_and_int& key_type;
  typedef boost::readable_property_map_tag category;
};

//get function for the property map
My_point_property_map::reference
get(My_point_property_map,My_point_property_map::key_type p)
{return boost::get<0>(p);}


typedef CGAL::Random_points_in_cube_3<Point_3>
                   Random_points_iterator;
typedef CGAL::Search_traits_3<Kernel>
                   Traits_base;
typedef
    CGAL::Search_traits_adapter<Point_and_int,My_point_property_map,Traits_base>
        Traits;


typedef CGAL::Orthogonal_k_neighbor_search<Traits>
    K_neighbor_search;
typedef K_neighbor_search::Tree
    Tree;
```

```
typedef K_neighbor_search::Distance
    Distance;

int main() {
  const unsigned int K = 5;
  // generator for random data points in the cube ( (-1,-1,-1), (1,1,1) )
  Random_points_iterator rpit( 1.0);
  std::vector<Point_3> points;
  std::vector<int>     indices;

  points.push_back(Point_3(*rpit++));
  points.push_back(Point_3(*rpit++));
  points.push_back(Point_3(*rpit++));
  points.push_back(Point_3(*rpit++));
  points.push_back(Point_3(*rpit++));
  points.push_back(Point_3(*rpit++));
  points.push_back(Point_3(*rpit++));

  indices.push_back(0);
  indices.push_back(1);
  indices.push_back(2);
  indices.push_back(3);
  indices.push_back(4);
  indices.push_back(5);
  indices.push_back(6);

  // Insert number_of_data_points in the tree
  Tree tree(
    boost::make_zip_iterator(boost::make_tuple(
   points.begin(),indices.begin() )),
    boost::make_zip_iterator(boost::make_tuple(
   points.end(),indices.end() ) )
  );
  Point_3 query(0.0, 0.0, 0.0);
  Distance tr_dist;

  // search K nearest neighbours
  K_neighbor_search search(tree, query, K);
  for(K_neighbor_search::iterator it = search.begin(); it !=
   search.end(); it++){
    std::cout << " d(q, nearest neighbor)=  "
              << tr_dist.inverse_of_transformed_distance(it->second) <<
   " " << boost::get<0>(it->first)<< " " << boost::get<1>(it->first) <<
   std::endl;
  }
  return 0;
}
```

File: examples/Spatial_searching/searching_with_point_with_info.cpp

Using an Integer as Key Type

In this example program, the search tree stores only integers that refer to points stored within a user vector. The point type of the search traits is *std::size_t*.

```
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/basic.h>
#include <CGAL/Search_traits_3.h>
#include <CGAL/Search_traits_adapter.h>
#include <CGAL/point_generators_3.h>
#include <CGAL/Orthogonal_k_neighbor_search.h>
#include <boost/iterator/counting_iterator.hpp>
#include <utility>

typedef CGAL::Exact_predicates_inexact_constructions_kernel Kernel;
typedef Kernel::Point_3 Point_3;

typedef std::size_t Point;

//definition of the property map and get
//function as friend function to have access to
//private member
class My_point_property_map{
  const std::vector<Point_3>& points;
public:
  typedef Point_3 value_type;
  typedef const value_type& reference;
  typedef Point key_type;
  typedef boost::readable_property_map_tag category;

  My_point_property_map(const std::vector<Point_3>& pts):points(pts){}

  friend reference get(const My_point_property_map& ppmap,key_type i)
  {return ppmap.points[i];}
};


typedef CGAL::Random_points_in_cube_3<Point_3>
                    Random_points_iterator;
typedef CGAL::Search_traits_3<Kernel>
                    Traits_base;
typedef
   CGAL::Search_traits_adapter<Point,My_point_property_map,Traits_base>
             Traits;


typedef CGAL::Orthogonal_k_neighbor_search<Traits>
   K_neighbor_search;
typedef K_neighbor_search::Tree
   Tree;
typedef Tree::Splitter
   Splitter;
typedef K_neighbor_search::Distance
   Distance;
```

```
int main() {
  const unsigned int K = 5;
  // generator for random data points in the cube ( (-1,-1,-1), (1,1,1) )
  Random_points_iterator rpit( 1.0);
  std::vector<Point_3> points;

  points.push_back(Point_3(*rpit++));
  points.push_back(Point_3(*rpit++));
  points.push_back(Point_3(*rpit++));
  points.push_back(Point_3(*rpit++));
  points.push_back(Point_3(*rpit++));
  points.push_back(Point_3(*rpit++));
  points.push_back(Point_3(*rpit++));

  My_point_property_map ppmap(points);

  // Insert number_of_data_points in the tree
  Tree tree(
    boost::counting_iterator<std::size_t>(0),
    boost::counting_iterator<std::size_t>(points.size()),
    Splitter(),
    Traits(ppmap)
  );
  Point_3 query(0.0, 0.0, 0.0);
  Distance tr_dist(ppmap);

  // search K nearest neighbours
  K_neighbor_search search(tree, query, K,0,true,tr_dist);
  for(K_neighbor_search::iterator it = search.begin(); it !=
   search.end(); it++){
    std::cout << " d(q, nearest neighbor)=  "
              << tr_dist.inverse_of_transformed_distance(it->second) <<
   " " << points[it->first] << " " << it->first << std::endl;
  }
  return 0;
}
```

File: examples/Spatial_searching/searching_with_point_with_info_inplace.cpp

Using a Model of L-value Property Map Concept

This example programs uses a BOOST LvaluePropertyMap. Points are read from a *std::map*. The search tree stores integers of type *std::size_t*. The value type of the iterator of the neighbor searching algorithm is *std::size_t*.

```
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/basic.h>
#include <CGAL/Search_traits_3.h>
#include <CGAL/Search_traits_adapter.h>
#include <CGAL/point_generators_3.h>
#include <CGAL/Orthogonal_k_neighbor_search.h>
#include <boost/iterator/counting_iterator.hpp>
```

```
#include <utility>

typedef CGAL::Exact_predicates_inexact_constructions_kernel Kernel;
typedef Kernel::Point_3 Point_3;

typedef std::size_t Point;

typedef boost::const_associative_property_map<std::map<Point,Point_3> >
                  My_point_property_map;

typedef CGAL::Random_points_in_cube_3<Point_3>
                  Random_points_iterator;
typedef CGAL::Search_traits_3<Kernel>
                  Traits_base;
typedef
   CGAL::Search_traits_adapter<Point,My_point_property_map,Traits_base>
             Traits;


typedef CGAL::Orthogonal_k_neighbor_search<Traits>
   K_neighbor_search;
typedef K_neighbor_search::Tree
   Tree;
typedef Tree::Splitter
   Splitter;
typedef K_neighbor_search::Distance
   Distance;

int main() {
  const unsigned int K = 5;
  // generator for random data points in the cube ( (-1,-1,-1), (1,1,1) )
  Random_points_iterator rpit( 1.0);
  std::map<Point,Point_3> points;

  points[0]=Point_3(*rpit++);
  points[1]=Point_3(*rpit++);
  points[2]=Point_3(*rpit++);
  points[3]=Point_3(*rpit++);
  points[4]=Point_3(*rpit++);
  points[5]=Point_3(*rpit++);
  points[6]=Point_3(*rpit++);

  My_point_property_map ppmap(points);

  // Insert number_of_data_points in the tree
  Tree tree(
    boost::counting_iterator<std::size_t>(0),
    boost::counting_iterator<std::size_t>(points.size()),
    Splitter(),
    Traits(ppmap)
  );
  Point_3 query(0.0, 0.0, 0.0);
  Distance tr_dist(ppmap);
```

```
  // search K nearest neighbours
  K_neighbor_search search(tree, query, K,0,true,tr_dist);
  for(K_neighbor_search::iterator it = search.begin(); it !=
   search.end(); it++){
    std::cout << " d(q, nearest neighbor)=  "
              << tr_dist.inverse_of_transformed_distance(it->second) <<
   " " << points[it->first] << " " << it->first << std::endl;
  }
  return 0;
}
```

File: examples/Spatial_searching/searching_with_point_with_info_pmap.cpp

61.3.7 Example for Selecting a Splitting Rule and Setting the Bucket Size

This example program illustrates selecting a splitting rule and setting the maximal allowed bucket size. The only differences with the first example are the declaration of the *Fair* splitting rule, needed to set the maximal allowed bucket size.

```
#include <CGAL/Simple_cartesian.h>
#include <CGAL/point_generators_2.h>
#include <CGAL/Search_traits_2.h>
#include <CGAL/Orthogonal_k_neighbor_search.h>
#include <cmath>

typedef CGAL::Simple_cartesian<double> R;
typedef R::Point_2 Point_d;
typedef CGAL::Random_points_in_square_2<Point_d> Random_points_iterator;
typedef CGAL::Counting_iterator<Random_points_iterator>
   N_Random_points_iterator;
typedef CGAL::Search_traits_2<R> Traits;
typedef CGAL::Euclidean_distance<Traits> Distance;
typedef CGAL::Fair<Traits> Fair;
typedef CGAL::Orthogonal_k_neighbor_search<Traits,Distance,Fair>
   Neighbor_search;
typedef Neighbor_search::Tree Tree;

int main() {
  const unsigned int N = 1000;
  // generator for random data points in the square ( (-1,-1), (1,1) )
  Random_points_iterator rpit( 1.0);

  Fair fair(5); // bucket size=5
  // Insert number_of_data_points in the tree
  Tree tree(N_Random_points_iterator(rpit,0),
            N_Random_points_iterator(N),
            fair);

  Point_d query(0,0);

  // Initialize the search structure, and search all N points
  Neighbor_search search(tree, query, N);

  // report the N nearest neighbors and their distance
  // This should sort all N points by increasing distance from origin
  for(Neighbor_search::iterator it = search.begin(); it != search.end();
   ++it){
    std::cout << it->first << " "<< std::sqrt(it->second) << std::endl;
  }
  return 0;
}
```

File: examples/Spatial_searching/using_fair_splitting_rule.cpp

61.4 Software Design

61.4.1 The *k*-*d* tree

Bentley [Ben75] introduced the *k*-*d* tree as a generalization of the binary search tree in higher dimensions. *k*-*d* trees hierarchically decompose space into a relatively small number of rectangles such that no rectangle contains too many input objects. For our purposes, a *rectangle* in real d dimensional space, $\mathbb{R}^d$, is the product of d closed intervals on the coordinate axes. *k*-*d* trees are obtained by partitioning point sets in $\mathbb{R}^d$ using (d-1)-dimensional hyperplanes. Each node in the tree is split into two children by one such separating hyperplane. Several splitting rules (see Section 61.2 can be used to compute a separating (d-1)-dimensional hyperplane.

Each internal node of the *k*-*d* tree is associated with a rectangle and a hyperplane orthogonal to one of the coordinate axis, which splits the rectangle into two parts. Therefore, such a hyperplane, defined by a splitting dimension and a splitting value, is called a separator. These two parts are then associated with the two child nodes in the tree. The process of partitioning space continues until the number of data points in the rectangle falls below some given threshold. The rectangles associated with the leaf nodes are called *buckets*, and they define a subdivision of the space into rectangles. Data points are only stored in the leaf nodes of the tree, not in the internal nodes.

Friedmann, Bentley and Finkel [FBF77] described the standard search algorithm to find the *k*th nearest neighbor by searching a *k*-*d* tree recursively.

When encountering a node of the tree, the algorithm first visits the child that is closest to the query point. On return, if the rectangle containing the other child lies within 1/ (1+ε) times the distance to the *k*th nearest neighbors so far, then the other child is visited recursively. Priority search [AM93b] visits the nodes in increasing order of distance from the queue with help of a priority queue. The search stops when the distance of the query point to the nearest nodes exceeds the distance to the nearest point found with a factor 1/ (1+ε). Priority search supports next neighbor search, standard search does not.

In order to speed-up the internal distance computations in nearest neighbor searching in high dimensional space, the approximate searching package supports orthogonal distance computation. Orthogonal distance computation implements the efficient incremental distance computation technique introduced by Arya and Mount [AM93a]. This technique works only for neighbor queries with query items represented as points and with a quadratic form distance, defined by $d_A(x,y) = (x-y)A(x-y)^T$, where the matrix A is positive definite, i.e. $d_A(x,y) \geq 0$. An important class of quadratic form distances are weighted Minkowski distances. Given a parameter $p > 0$ and parameters $w_i \geq 0, 0 < i \leq d$, the weighted Minkowski distance is defined by $l_p(w)(r,q) = (\Sigma_{i=1}^{i=d} w_i(r_i - q_i)^p)^{1/p}$ for $0 < p < \infty$ and defined by $l_\infty(w)(r,q) = max\{w_i|r_i - q_i| \mid 1 \leq i \leq d\}$. The Manhattan distance ($p = 1$, $w_i = 1$) and the Euclidean distance ($p = 2$, $w_i = 1$) are examples of a weighted Minkowski metric.

To speed up distance computations also transformed distances are used instead of the distance itself. For instance for the Euclidean distance, to avoid the expensive computation of square roots, squared distances are used instead of the Euclidean distance itself.

dD Spatial Searching Reference Manual

Hans Tangelder and Andreas Fabri

This package provides data structures and algorithms for exact and approximate distance browsing, supporting

- both nearest and furthest neighbor searching,
- both exact and approximate searching,
- (approximate) range searching,
- (approximate) *k*-nearest and *k*-furthest neighbor searching,
- (approximate) incremental nearest and incremental furthest neighbor searching,
- query items representing points and spatial objects.

The spatial searching package consists of the following concepts and classes that are described in the reference pages.

61.5 Classified Reference Pages

Search Classes

Range Query Item Classes

Search Traits Classes

Distance Classes

Splitter Classes

Advanced Classes

Concepts

61.6 Alphabetical List of Reference Pages

Class

CGAL::Euclidean_distance<Traits>

Definition

The class *Euclidean_distance<Traits>* provides an implementation of the concept *OrthogonalDistance*, with the Euclidean distance (l_2 metric). To optimize distance computations squared distances are used.

#include <CGAL/Euclidean_distance.h>

Parameters

Expects for the first template argument a model of the concept *SearchTraits*, for example *CGAL::Search_traits_2<CGAL::Cartesian<double> >*.

Is Model for the Concepts

OrthogonalDistance

Types

Traits::FT	*FT;*	Number type.
Traits::Point_d	*Point_d;*	Point type.
Point_d	*Query_item;*	Query item type.

Creation

Euclidean_distance<Traits> ed(Traits t=Traits());

Default constructor.

Operations

FT *ed.transformed_distance(Query_item q, Point_d p) const*

Returns the squared Euclidean distance between q and p.

FT *ed.min_distance_to_rectangle(Query_item q, Kd_tree_rectangle<FT> r) const*

Returns the squared Euclidean distance between q and the point on the boundary of r closest to q.

FT	*ed.max_distance_to_rectangle(Query_item q, Kd_tree_rectangle<FT> r;) const*	
		Returns the squared Euclidean distance, where d denotes the distance between q and the point on the boundary of r farthest to q.
FT	*ed.new_distance(FT dist, FT old_off, FT new_off, int cutting_dimension) const*	
		Updates the squared *dist* incrementally and returns the updated squared distance.
FT	*ed.transformed_distance(FT d) const*	
		Returns d^2.
FT	*ed.inverse_of_transformed_distance(FT d) const*	
		Returns $d^{1/2}$.

See Also

OrthogonalDistance
CGAL::Weighted_Minkowski_distance<Traits>

Class

CGAL::Euclidean_distance_sphere_point<Traits>

Definition

The class *Euclidean_distance_sphere_point<Traits>* provides an implementation of the *GeneralDistance* concept for the Euclidean distance (l_2 metric) between a d-dimensional sphere and a point, and the Euclidean distance between a d-dimensional sphere and a d-dimensional iso-rectangle defined as a k-d tree rectangle.

#include <CGAL/Euclidean_distance_sphere_point.h>

Parameters

Expects for the template argument a model of the concept *SearchTraits*, for example *CGAL::Cartesian_d< double>*.

Is Model for the Concepts

GeneralDistance

Types

Traits::FT	*FT;*	Number type.
Traits::Point_d	*Point_d;*	Point type.
Traits::Sphere_d	*Sphere_d;*	Query item type.

Creation

Euclidean_distance_sphere_point<Traits> ed(Traits t=Traits());

Default constructor.

Operations

NT *ed.transformed_distance(Query_item s, Point_d p) const*

Returns the distance between s and p.

NT *ed.min_distance_to_rectangle(Query_item s, Kd_tree_rectangle<FT> r) const*

Returns the minimal distance between a point from the sphere s and a point from r.

NT *ed.max_distance_to_rectangle(Query_item s, Kd_tree_rectangle<FT> r) const*

Returns the maximal distance between the sphere *s* and a point from *r* furthest to *s*.

NT *ed.transformed_distance(NT d) const*

Returns d^2.

NT *ed.inverse_of_transformed_distance(NT d) const*

Returns $d^{1/2}$.

See Also

GeneralDistance

Functor
Class

CGAL::Fair<Traits, SpatialSeparator>

Definition

Implements the *fair* splitting rule. This splitting rule is a compromise between the median of rectangle splitting rule and the midpoint of rectangle splitting rule. This splitting rule maintains an upper bound on the maximal allowed ratio of the longest and shortest side of a rectangle (the value of this upper bound is set in the constructor of the fair splitting rule). Among the splits that satisfy this bound, it selects the one in which the points have the largest spread. It then splits the points in the most even manner possible, subject to maintaining the bound on the ratio of the resulting rectangles.

#include <CGAL/Splitters.h>

Parameters

Expects for the first template argument a model of the concept *SearchTraits*, for example *Cartesian_d<double>*.

Expects for the second template argument a model of the concept *Separator*. It has as default value the type, *CGAL::Plane_separator<Traits::FT>*.

Is Model for the Concepts

Splitter

Types

Traits::FT	*FT;*	Number type.

Creation

Fair<Traits, SpatialSeparator> s;	Default constructor.

Fair<Traits, SpatialSeparator> s(unsigned int bucket_size, FT aspect_ratio=FT(3));

Constructor.

See Also

Splitter
SpatialSeparator

FuzzyQueryItem

Concept

Definition

The concept FuzzyQueryItem describes the requirements for fuzzy d-dimensional spatial objects.

Has Models

CGAL::Fuzzy_sphere<Traits>,
CGAL::Fuzzy_iso_box<Traits>

Types

FuzzyQueryItem:: Point_d	represents a d-dimensional point.
FuzzyQueryItem:: FT	Number type.

Operations

bool *q.contains(Point_d p) const*

test whether q contains p.

bool *q.inner_range_intersects(Kd_tree_rectangle<FT> rectangle) const*

test whether the inner approximation of the spatial object intersects a rectangle associated with a node of a tree.

bool *q.outer_range_contains(Kd_tree_rectangle<FT> rectangle) const*

test whether the outer approximation of the spatial object encloses the rectangle associated with a node of a tree.

Class

CGAL::Fuzzy_iso_box<Traits>

Definition

The class *Fuzzy_iso_box<Traits>* implements fuzzy d-dimensional iso boxes. A fuzzy iso box with fuzziness value ε has as outer approximation a box dilated, and as inner approximation a box eroded by a d-dim square with side length ε.

#include <CGAL/Fuzzy_iso_box.h>

Parameters

Expects for the template argument a model of the concept *SearchTraits*, for example *CGAL::Search_traits_2< CGAL::Simple_cartesian<double> >*.

Is Model for the Concepts

FuzzyQueryItem

Types

Traits::Point_d	*Point_d;*	Point type.
Traits::FT	*FT;*	Number type.

Creation

Fuzzy_iso_box<Traits> b(Point_d p, Point_d q, FT epsilon=FT(0), Traits t=Traits());

Constructs a fuzzy iso box specified by the minimal iso box containing p and q and fuzziness value *epsilon*.
Precondition: p must be lexicographically smaller than q.

In case *Traits* is *Search_traits_adapter<Key,PointPropertyMap,BaseTraits>*, the following constructor is also available:

Fuzzy_iso_box<Traits> b(Traits::Base::Point_d p,
Traits::Base::Point_d q,
FT epsilon=FT(0),
Traits t=Traits())

Constructs a fuzzy iso box specified by the minimal iso box containing p and q and fuzziness value *epsilon*.
Precondition: p must be lexicographically smaller than q.

Operations

bool *b.contains(Point_d p) const*

test whether *b* contains *p*.

bool *b.inner_range_intersects(Kd_tree_rectangle<FT> rectangle) const*

test whether the inner box intersects the rectangle associated with a node of a tree.

bool *b.outer_range_contains(Kd_tree_rectangle<FT> rectangle) const*

test whether the outer box encloses the rectangle associated with a node of a tree.

See Also

FuzzyQueryItem

Class

CGAL::Fuzzy_sphere<Traits>

Definition

The class *Fuzzy_sphere<Traits>* implements fuzzy d-dimensional spheres. A fuzzy sphere with radius r and fuzziness value ε has as outer approximation a sphere with radius $r+\varepsilon$ and as inner approximation a sphere with radius $r-\varepsilon$.

#include <CGAL/Fuzzy_sphere.h>

Parameters

Expects for the template argument a model of the concept *SearchTraits*, for example *CGAL::Cartesian_d< double>*.

Is Model for the Concepts

FuzzyQueryItem

Types

Traits::Point_d	*Point_d;*	Point type.
Traits::FT	*FT;*	Number type.

Creation

Fuzzy_sphere<Traits> s(Point_d center, FT radius, FT epsilon=FT(0), Traits t=Traits());

Constructs a fuzzy sphere centered at *center* with radius *radius* and fuzziness value *epsilon*.

In case *Traits* is *Search_traits_adapter<Key,PointPropertyMap,BaseTraits>*, the following constructor is also available:

Fuzzy_sphere<Traits> s(Traits::Base::Point_d center, FT radius, FT epsilon=FT(0), Traits t=Traits());

Constructs a fuzzy sphere centered at *center* with radius *radius* and fuzziness value *epsilon*.

Operations

bool *s.contains(Point_d p) const*

test whether *s* contains *p*.

bool *s.inner_range_intersects(Kd_tree_rectangle<FT> rectangle) const*

test whether the inner sphere intersects the rectangle associated with a node of a tree.

bool *s.outer_range_contains(Kd_tree_rectangle<FT> rectangle) const*

test whether the outer sphere encloses the rectangle associated with a node of a tree.

See Also

FuzzyQueryItem

Concept

GeneralDistance

Definition

Requirements of a distance class defining a distance between a query item denoting a spatial object and a point. To optimize distance computations transformed distances are used, e.g., for a Euclidean distance the transformed distance is the squared Euclidean distance.

Has Models

CGAL::Manhattan_distance_iso_box_point<Traits>
CGAL::Euclidean_distance_sphere_point<Traits>.

Types

GeneralDistance:: FT	Number type.
GeneralDistance:: Point_d	Point type.
GeneralDistance:: Query_item	Query item type.

Operations

FT *gd.transformed_distance(Query_item q, Point_d r)*

Returns the transformed distance between *q* and *r*.

FT *gd.min_distance_to_rectangle(Query_item q, Kd_tree_rectangle<FT> r) const*

Returns the transformed distance between *q* and the point on the boundary of *r* closest to *q*.

FT *gd.max_distance_to_rectangle(Query_item q, Kd_tree_rectangle<FT> r) const*

Returns the transformed distance between *q* and the point on the boundary of *r* furthest to *q*.

FT *gd.transformed_distance(FT d) const*

Returns the transformed distance.

FT *gd.inverse_of_transformed_distance(FT d) const*

Returns the inverse of the transformed distance.

CGAL::Incremental_neighbor_search<Traits, GeneralDistance, Splitter, SpatialTree>

Class

Definition

The class *Incremental_neighbor_search<Traits, GeneralDistance, Splitter, SpatialTree>* implements incremental nearest and furthest neighbor searching on a tree. The tree may have extended or unextended nodes.

#include <CGAL/Incremental_neighbor_search.h>

Parameters

Expects for the first template argument a model of the concept *SearchTraits*, for example *CGAL::Search_traits_2<CGAL::Cartesian<double> >*.

Expects for the second template argument a model of the concept *GeneralDistance*. If *Traits* is *CGAL::Search_traits_adapter<Key,PointPropertyMap,BaseTraits>* the default type is *CGAL::Distance_for_point_adapter<Key,PointPropertyMap,CGAL::Euclidean_distance<Traits> >*, and *CGAL::Euclidean_distance<Traits>* otherwise.

Expects for third template argument a model of the concept *Splitter*. The default type is *CGAL::Sliding_midpoint<Traits>*.

Expects for fourth template argument a model of the concept *SpatialTree*. The default type is *CGAL::Kd_tree<Traits,Splitter,CGAL::Tag_false>*. The template argument *CGAL::Tag_false* makes that the tree is built with unextended nodes.

Types

Traits::Point_d	*Point_d;*	Point type.
Traits::NT	*NT;*	Number type.
GeneralDistance	*Distance;*	Distance type.
std::pair<Point_d,NT>	*Point_with_transformed_distance;*	Pair of point and transformed distance.

Incremental_neighbor_search<Traits, GeneralDistance, Splitter, SpatialTree>:: iterator

const input iterator with value type *Point_with_transformed_distance* for enumerating approximate neighbors.

GeneralDistance::Query_item

	Query_item;	Query item type.
SpatialTree	*Tree;*	The tree type.

Creation

Incremental_neighbor_search<Traits, GeneralDistance, Splitter, SpatialTree> s(
Tree& tree,
QueryItem q,
NT eps=NT(0.0),
bool search_nearest=true,
GeneralDistance d=GeneralDistance())

Constructor for incremental neighbor searching of the query item *q* in the points stored *tree* using a distance *d* and approximation factor *eps*.

Operations

iterator	*s.begin() const*	Returns a const iterator to the approximate nearest or furthest neighbor.
iterator	*s.end() const*	Returns the appropriate past-the-end const iterator.

advanced

std::ostream& *s.statistics(std::ostream& s)*

Inserts statistics of the search process into the output stream *s*.

advanced

See Also

CGAL::Orthogonal_incremental_neighbor_search<Traits, OrthogonalDistance, Splitter, SpatialTree>

CGAL::K_neighbor_search<Traits, GeneralDistance, Splitter, SpatialTree>

Class

Definition

The class *K_neighbor_search<Traits, GeneralDistance, Splitter, SpatialTree>* implements approximate k-nearest and k-furthest neighbor searching using standard search on a tree using a general distance class. The tree may be built with extended or unextended nodes.

#include <CGAL/K_neighbor_search.h>

Parameters

Expects for the first template argument an implementation of the concept *SearchTraits*, for example *CGAL::Cartesian_d<double>*.

Expects for the second template argument a model of the concept *GeneralDistance*. If *Traits* is *CGAL::Search_traits_adapter<Key,PointPropertyMap,BaseTraits>* the default type is *CGAL::Distance_for_point_adapter< Key,PointPropertyMap,CGAL::Euclidean_distance<Traits> >*, and *CGAL::Euclidean_distance<Traits>* otherwise.

Expects for fourth template argument an implementation of the concept *SpatialTree*. The default type is *CGAL::Kd_tree<Traits, Splitter, CGAL::Tag_false>*. The template argument *CGAL::Tag_false* makes that the tree is built with unextended nodes.

Types

Traits::Point_d	*Point_d;*	Point type.
Traits::FT	*FT;*	Number type.
GeneralDistance	*Distance;*	Distance type.
std::pair<Point_d,FT>	*Point_with_transformed_distance;*	Pair of point and transformed distance.

K_neighbor_search<Traits, GeneralDistance, Splitter, SpatialTree>:: iterator

Bidirectional const iterator with value type *Point_with_distance* for enumerating approximate neighbors.

GeneralDistance::Query_item

Query_item; Query item type.

SpatialTree	*Tree;*	The tree type.

Creation

K_neighbor_search<Traits, GeneralDistance, Splitter, SpatialTree> s(
Tree tree,
Query_item q,
unsigned int k=1,
FT eps=FT(0.0),
bool search_nearest=true,
GeneralDistance d=GeneralDistance(),
bool sorted=true)

Constructor for searching approximately k neighbors of the query item q in the points stored in *tree* using distance class d and approximation factor *eps*. *sorted* indicates if the computed sequence of k-nearest neighbors needs to be sorted.

Operations

iterator	*s.begin() const*	Returns a const iterator to the approximate nearest or furthest neighbor.
iterator	*s.end() const*	Returns the appropriate past-the-end const iterator.

advanced

std::ostream& *s.statistics(std::ostream& s)*

Inserts statistics of the search process into the output stream s.

advanced

See Also

CGAL::Orthogonal_k_neighbor_search<Traits, OrthogonalDistance, Splitter, SpatialTree>

s

Class

CGAL::Kd_tree<Traits, Splitter, UseExtendedNode>

Definition

The class *Kd_tree<Traits, Splitter, UseExtendedNode>* defines a *k-d* tree.

#include <CGAL/Kd_tree.h>

Parameters

Expects for the first template argument a model of the concept *SearchTraits*, for example *CGAL::Search_traits_2<CGAL::Cartesian<double> >*.

Expects for the second template argument a model for the concept *Splitter*. It defaults to *Sliding_midpoint< Traits>*.

Expects for the third template argument *CGAL::Tag_true*, if the tree shall be built with extended nodes, and *CGAL::Tag_false* otherwise.

Types

Traits::Point_d	*Point_d;*	Point class.
Traits::FT	*FT;*	Number type.

Kd_tree<Traits, Splitter, UseExtendedNode>:: Splitter

Splitter type.

Kd_tree<Traits, Splitter, UseExtendedNode>:: iterator

Bidirectional const iterator with value type *Point_d* that allows to enumerate all points in the tree.

advanced

Kd_tree<Traits, Splitter, UseExtendedNode>:: Node_handle

A handle with value type *Kd_tree_node<Traits,Splitter>*.

Kd_tree<Traits, Splitter, UseExtendedNode>:: Node_const_handle

A const handle with value type *Kd_tree_node< Traits,Splitter>*.

Kd_tree<Traits, Splitter, UseExtendedNode>:: Point_d_iterator

Random access const iterator with value type *const Point_d**.

Kd_tree<Traits, Splitter, UseExtendedNode>:: size_type

A type that counts the number of elements in a *k-d* tree.

advanced

Creation

Kd_tree<Traits, Splitter, UseExtendedNode> tree(Splitter s=Splitter(), Traits t=Traits());

Constructs an empty *k-d* tree.

template <class InputIterator>
Kd_tree<Traits, Splitter, UseExtendedNode> tree(InputIterator first, InputIterator beyond, Splitter s=Splitter(), Traits t=Traits())

Constructs a *k-d* tree on the elements from the sequence *[first, beyond)* using the splitting rule implemented by *s*. The value type of the *InputIterator* must be *Point_d*.

Operations

void *tree.insert(Point_d p)*

Inserts the point *p* in the *k-d* tree.

template <class InputIterator>
void *tree.insert(InputIterator first, InputIterator beyond)*

Inserts the elements from the sequence *[first, beyond)* in the *k-d* tree. The value type of the *InputIterator* must be *Point_d*.

template <class OutputIterator, class FuzzyQueryItem>
OutputIterator *tree.search(OutputIterator it, FuzzyQueryItem q) const*

Reports the points that are approximately contained by *q*. The types *FuzzyQueryItem::Point_d* and *Point_d* must be equivalent. To use this function *Traits* must be a model of the concept *RangeSearchTraits*.

iterator *tree.begin() const* Returns a const iterator to the first point in the tree.

iterator *tree.end() const* Returns the appropriate past-the-end const iterator.

void *tree.clear()* Removes all points from the *k-d* tree.

size_type	*tree.size() const*	Returns the number of points that are stored in the tree.
Traits	*tree.traits() const*	return the instance of the traits used to construct the tree.

advanced

Node_handle	*tree.root()*	Returns a handle to the root node of the tree.
Node_const_handle	*tree.root() const*	Returns a const handle to the root node of the tree.
Kd_tree_rectangle<FT>	*tree.bounding_box() const*	returns a const reference to the bounding box of the root node of the tree.
std::ostream&	*tree.statistics(std::ostream& s) const*	Inserts statistics of the tree into the output stream *s*.

advanced

See Also

Tree. *CGAL::Kd_tree_node<Traits>*
CGAL::Search_traits_2<Kernel>
CGAL::Search_traits_3<Kernel>
CGAL::Search_traits<FT_,Point,CartesianIterator,ConstructCartesianIterator>

CGAL::Kd_tree_node<Traits, Splitter, UseExtendedNode>

advanced

Definition

The class *Kd_tree_node<Traits, Splitter, UseExtendedNode>* implements a node class for a *k-d* tree. A node is either a leaf node, an internal node or an extended internal node. A leaf node contains one or more points. An internal node contains a pointer to its lower child, a pointer to its upper child, and a pointer to its separator. An extended internal node is an internal node containing the lower and upper limit of an extended node's rectangle along the node's cutting dimension.

#include <CGAL/Kd_tree_node.h>

Parameters

Expects for the template argument a model of the concept *SearchTraits*, for example *CGAL::Search_traits_2< CGAL::Cartesian<double> >*, or *CGAL::Cartesian_d<double>*.

Types

enum Node_type { *LEAF, INTERNAL, EXTENDED_INTERNAL*}*;*

Denotes type of node.

Traits::FT	*FT;*	Number type.
Traits::Point_d	*Point_d;*	Point type.
Splitter::Separator	*Separator;*	Separator type.

Kd_tree<Traits,Splitter,UseExtendedNode>::Point_d_iterator

Point_d_iterator; const iterator over points.

Kd_tree<Traits,Splitter,UseExtendedNode>::Node_handle

Node_handle; Node handle.

Kd_tree<Traits,Splitter,UseExtendedNode>::Node_const_handle

Node_const_handle; const node handle.

Creation

Operations

template <class OutputIterator, class FuzzyQueryItem>
OutputIterator *n.search(OutputIterator it, FuzzyQueryItem q) const*

Reports the points from the subtree of the node, that are approximately contained by q.

template <class OutputIterator>
OutputIterator *n.tree_items(OutputIterator it) const*

Reports all the points contained by the subtree of the node.

bool	*n.is_leaf() const*	Indicates whether a node is a leaf node.
unsigned int	*n.size() const*	Returns the number of items stored in a leaf node.
Point_d_iterator	*n.begin() const*	Returns a const iterator to the first item in a leaf node.
Point_d_iterator	*n.end() const*	Returns the appropriate past-the-end const iterator.
Node_handle	*n.lower()*	Returns a handle to the lower child of an internal node.
Node_handle	*n.upper()*	Returns a handle to the upper child of an internal node.
Node_const_handle	*n.lower() const*	Returns a const handle to the lower child of an internal node.
Node_const_handle	*n.upper() const*	Returns a const handle to the upper child of an internal node.
Separator&	*n.separator()*	Returns a reference to the separator.
FT	*n.low_value() const*	Returns the lower limit of an extended node's rectangle along the node's cutting dimension.
FT	*n.high_value() const*	Returns the upper limit of an extended node's rectangle along the node's cutting dimension.

advanced

CGAL::Kd_tree_rectangle<FT>

advanced

Definition

The class *Kd_tree_rectangle<FT>* implements d-dimensional iso-rectangles and related operations, e.g., methods to compute bounding boxes of point sets.

#include <CGAL/Kd_tree_rectangle.h>

Types

FT	*FT;*	Number type.

Creation

Kd_tree_rectangle<FT>	*r(int d);*	Constructs a d-dimensional rectangle r with lower bound and upper bound set to zero in each dimension.

template <class Construct_cartesian_const_iterator_d,class PointIter>
Kd_tree_rectangle<FT> r(int d, PointIter begin, PointIter end, Construct_cartesian_const_iterator_d construct_it)

Constructs the bounding box of the points in the range [*begin,end*), where the value type of *PointIter* can be used by operators of functors *Construct_cartesian_const_iterator_d* to define iterators with value type *FT*.

Operations

FT	*r.min_coord(int i) const*	
		Returns the lower bound of the rectangle in dimension i.
FT	*r.max_coord(int i) const*	
		Returns the upper bound of the rectangle in dimension i.
void	*r.set_upper_bound(int i, FT x)*	
		Sets upper bound in dimension i to x.

Class

void	*r.set_lower_bound(int i, FT x)*	
		Sets lower bound in dimension *i* to *x*.
FT	*r.max_span() const*	Returns the maximal span of the rectangle.
FT	*r.max_span_coord() const*	
		Returns the smallest coordinate for which the rectangle has its maximal span.
int	*r.dimension() const*	Returns the dimension of the rectangle.
void	*r.split(& r, int d, FT value)*	
		Splits rectangle in dimension *d* at coordinate-value *value* by modifying itself to lower half and by modifying *r* to upper half.

Output Routines

template<class FT> *std::ostream&*	*std::ostream& s << & r*	
		Inserts rectangle *r* in the output stream *s* and returns *s*.

advanced

CGAL::Manhattan_distance_iso_box_point<Traits>

Definition

The class *Manhattan_distance_iso_box_point<Traits>* provides an implementation of the *GeneralDistance* concept for the Manhattan distance (l_1 metric) between a d-dimensional iso-box and a d-dimensional point and the Manhattan distance between a d-dimensional iso-box and a d-dimensional iso-box defined as a k-d tree rectangle.

#include <CGAL/Manhattan_distance_iso_box_point.h>

Parameters

Expects for the template argument a model for the concept *SearchTraits*, for example *CGAL::Search_traits_3< CGAL::Cartesian<double> >*.

Is Model for the Concepts

GeneralDistance

Types

Traits::FT	*FT;*	Number type.
Traits::Point_d	*Point_d;*	Point type.
Traits::Iso_box_d	*Query_item;*	Query item type.

Creation

Manhattan_distance_iso_box_point<Traits> md(Traits t=Traits());

Default constructor.

Operations

FT *md.transformed_distance(Query_item b, Point_d p) const*

Returns the transformed distance between b and p.

FT *md.transformed_distance(FT d) const*

Returns the transformed value of of d.

FT *md.inverse_of_transformed_distance(FT d) const*

Returns the value of the inverse of the transform function applied to d.

FT *md.min_distance_to_rectangle(Query_item b, Kd_tree_rectangle<FT> r) const*

Returns the minimal distance between a point from b and a point from r.

FT *md.max_distance_to_rectangle(Query_item b, Kd_tree_rectangle<FT> r) const*

Returns the maximal distance between the iso-box b and a point from r furthest to b.

See Also

GeneralDistance

Functor
Class

CGAL::Median_of_max_spread<Traits, SpatialSeparator>

Definition

Implements the *median of max spread* splitting rule. The splitting dimension is the dimension of the longest side of the rectangle. The splitting value is defined by the median of the coordinates of the data points along this dimension.

#include <CGAL/Splitters.h>

Parameters

Expects for the first template argument a model of the concept *SearchTraits*, for example the type *CGAL::Search_traits_3< Cartesian<double> >*.

Expects for the second template argument a model of the concept *Separator*. It has as default value the type, *CGAL::Plane_separator<Traits::FT>*.

Is Model for the Concepts

Splitter

Creation

Median_of_max_spread<Traits, SpatialSeparator> s;

Default constructor.

Median_of_max_spread<Traits, SpatialSeparator> s(unsigned int bucket_size);

Constructor.

See Also

Splitter
SpatialSeparator

Functor
Class

CGAL::Median_of_rectangle<Traits, SpatialSeparator>

Definition

Implements the *median of rectangle* splitting rule. The splitting dimension is the dimension of the longest side of the rectangle. The splitting value is defined by the median of the coordinates of the data points along this dimension.

#include <CGAL/Splitters.h>

Parameters

Expects for the first template argument a model of the concept *SearchTraits*, for example the type *CGAL::Search_traits_3< Cartesian<double> >*.

Expects for the second template argument a model of the concept *Separator*. It has as default value the type, *CGAL::Plane_separator<Traits::FT>*.

Is Model for the Concepts

Splitter

Creation

Median_of_rectangle<Traits, SpatialSeparator> s;

Default constructor.

Median_of_rectangle<Traits, SpatialSeparator> s(unsigned int bucket_size);

Constructor.

See Also

Splitter,
SpatialSeparator

CGAL::Midpoint_of_max_spread<Traits, SpatialSeparator>

Functor
Class

Definition

Implements the *midpoint of max spread* splitting rule. A rectangle is cut through $(Mind + Maxd)/2$ orthogonal to the dimension with the maximum point spread $[Mind, Maxd]$.

#include <CGAL/Splitters.h>

Parameters

Expects for the first template argument a model of the concept *SearchTraits*, for example the type *CGAL::Search_traits_3< Cartesian<double> >*.

Expects for the second template argument a model of the concept *Separator*. It has as default value the type, *CGAL::Plane_separator<Traits::FT>*

Is Model for the Concepts

Splitter

Creation

Midpoint_of_max_spread<Traits, SpatialSeparator> s;

Default constructor.

Midpoint_of_max_spread<Traits, SpatialSeparator> s(unsigned int bucket_size);

Constructor.

See Also

Splitter,
SpatialSeparator

Functor
Class

CGAL::Midpoint_of_rectangle<Traits, SpatialSeparator>

Definition

Implements the *midpoint of rectangle* splitting rule. A rectangles is cut through its midpoint orthogonal to the longest side.

#include <CGAL/Splitters.h>

Parameters

Expects for the first template argument a model of the concept *SearchTraits*, for example the type *CGAL::Search_traits_3< Cartesian<double> >*.

Expects for the second template argument a model of the concept *Separator*. It has as default value the type, *CGAL::Plane_separator<Traits::FT>*

Is Model for the Concepts

Splitter

Creation

Midpoint_of_rectangle<Traits, SpatialSeparator> s;

Default constructor.

Midpoint_of_rectangle<Traits, SpatialSeparator> s(unsigned int bucket_size);

Constructor.

See Also

Splitter
SpatialSeparator

OrthogonalDistance

Concept

Definition

Requirements of an orthogonal distance class supporting incremental distance updates. To optimize distance computations transformed distances are used. E.g., for an Euclidean distance the transformed distance is the squared Euclidean distance.

Refines

GeneralDistance

Has Models

CGAL::Euclidean_distance<Traits>,
CGAL::Weighted_Minkowski_distance<Traits>

Types

OrthogonalDistance:: FT	Number type.
OrthogonalDistance:: Point_d	Point type.
OrthogonalDistance:: Query_item	Query item type.

Creation

OrthogonalDistance od(int d);	Constructor implementing distance for d-dimensional points.

Operations

FT *od.transformed_distance(Query_item q, Point_d r) const*

Returns the transformed distance between q and r.

FT *od.min_distance_to_rectangle(Query_item q, Kd_tree_rectangle<FT> r) const*

Returns the transformed distance between q and the point on the boundary of r closest to q.

FT *od.max_distance_to_rectangle(Query_item q, Kd_tree_rectangle<FT> r) const*

Returns the transformed distance between q and the point on the boundary of r farthest to q.

FT *od.transformed_distance(FT d) const*

Returns the transformed distance.

FT *od.inverse_of_transformed_distance(FT d) const*

Returns the inverse of the transformed distance.

FT *od.new_distance(FT dist, FT old_off, FT new_off, int cutting_dimension) const*

Updates *dist* incrementally and returns the updated distance.

CGAL::Orthogonal_incremental_neighbor_search<Traits, OrthogonalDistance, Splitter, SpatialTree>

Class

Definition

The class *Orthogonal_incremental_neighbor_search<Traits, OrthogonalDistance, Splitter, SpatialTree>* implements incremental nearest and furthest neighbor searching on a tree.

#include <CGAL/Orthogonal_incremental_neighbor_search.h>

Parameters

Expects for the first template argument a model of the concept *SearchTraits*, for example *CGAL::Search_traits_2<CGAL::Cartesian<double> >*.

Expects for the second template argument a model of the concept *GeneralDistance*. If *Traits* is *CGAL::Search_traits_adapter<Key,PointPropertyMap,BaseTraits>* the default type is *CGAL::Distance_for_point_adapter<Key,PointPropertyMap,CGAL::Euclidean_distance<Traits> >*, and *CGAL::Euclidean_distance<Traits>* otherwise.

Expects for third template argument a model of the concept *Splitter*. The default type is *CGAL::Sliding_midpoint<Traits>*.

Expects for fourth template argument a model of the concept *SpatialTree*. The default type is *CGAL::Kd_tree<Traits, Splitter, CGAL::Tag_true>*. The template argument must be *CGAL::Tag_true* because orthogonal search needs extended kd tree nodes.

Types

Traits::Point_d	*Point_d;*	Point type.
Traits::FT	*FT;*	Number type.
OrthogonalDistance	*Distance;*	Distance type.
OrthogonalDistance::Query_item	*Query_item;*	Query item.
std::pair<Point_d,FT>	*Point_with_transformed_distance;*	Pair of point and transformed distance.

Orthogonal_incremental_neighbor_search<Traits, OrthogonalDistance, Splitter, SpatialTree>:: iterator

const input iterator with value type *Point_with_transformed_distance* for enumerating approximate neighbors.

SpatialTree	*Tree;*	The tree type.

Creation

Orthogonal_incremental_neighbor_search<Traits, OrthogonalDistance, Splitter, SpatialTree> s(
SpatialTree& tree,
Query_item query,
FT eps=FT(0.0),
bool search_nearest=true,
OrthogonalDistance d=OrthogonalDistance())

Constructor for incremental neighbor searching of the query item *query* in the points stored *tree* using a distance *d* and approximation factor *eps*.

Operations

iterator	*s.begin() const*	Returns a const iterator to the approximate nearest or furthest neighbor.
iterator	*s.end() const*	Returns the appropriate past-the-end const iterator.

advanced

std::ostream& *s.statistics(std::ostream& s) const*

Inserts statistics of the search process into the output stream *s*.

advanced

See Also

CGAL::Incremental_neighbor_search<Traits, GeneralDistance, SpatialTree>

CGAL::Orthogonal_k_neighbor_search<Traits, OrthogonalDistance, Splitter, SpatialTree>

Class

Definition

The class *Orthogonal_k_neighbor_search<Traits, OrthogonalDistance, Splitter, SpatialTree>* implements approximate k-nearest and k-furthest neighbor searching on a tree using an orthogonal distance class.

#include <CGAL/Orthogonal_k_neighbor_search.h>

Parameters

Expects for the first template argument an implementation of the concept *SearchTraits*, for example *CGAL::Search_traits_2<CGAL::Cartesian<double> >*.

Expects for the second template argument a model of the concept *GeneralDistance*. If *Traits* is *CGAL::Search_traits_adapter<Key,PointPropertyMap,BaseTraits>* the default type is *CGAL::Distance_adapter<Key,PointPropertyMap,CGAL::Euclidean_distance<Traits> >*, and *CGAL::Euclidean_distance<Traits>* otherwise.

The default type is *CGAL::Euclidean_distance<Traits>*.

Expects for third template argument a model of the concept *Splitter*. The default type is *CGAL::Sliding_midpoint<Traits>*.

Expects for fourth template argument an implementation of the concept *SpatialTree*. The default type is *CGAL::Kd_tree<Traits, Splitter, CGAL::Tag_true>*. The template argument must be *CGAL::Tag_true* because orthogonal search needs extended kd tree nodes.

Types

Traits::Point_d	*Point_d;*	Point type.
Traits::FT	*FT;*	Number type.
OrthogonalDistance	*Distance;*	Distance type.
GeneralDistance::Query_item	*Query_item;*	Query item.
std::pair<Point_d,FT>	*Point_with_transformed_distance;*	Pair of point and transformed distance.

Orthogonal_k_neighbor_search<Traits, OrthogonalDistance, Splitter, SpatialTree>:: iterator

Bidirectional const iterator with value type *Point_with_transformed_distance* for enumerating approximate neighbors.

SpatialTree	*Tree;*	The tree type.

Operations

Orthogonal_k_neighbor_search<Traits, OrthogonalDistance, Splitter, SpatialTree> s(
SpatialTree tree,
Query_item query,
unsigned int k=1,
FT eps=FT(0.0),
bool search_nearest=true,
OrthogonalDistance d=OrthogonalDistance(),
bool sorted=true)

Constructor for searching approximately *k* neighbors of the query item *query* in the points stored in *tree* using distance *d* and approximation factor *eps*.*sorted* indicates if the computed sequence of *k*-nearest neighbors needs to be sorted.

iterator	*s.begin() const*	Returns a const iterator to the approximate nearest or furthest neighbor.
iterator	*s.end() const*	Returns the appropriate past-the-end const iterator.

advanced

std::ostream& *s.statistics(std::ostream& s)*

Inserts statistics of the search process into the output stream *s*.

advanced

See Also

CGAL::K_neighbor_search<Traits, GeneralDistance, Splitter, SpatialTree>

CGAL::Plane_separator<FT>

advanced

Definition

The class *Plane_separator<FT>* implements a plane separator, i.e., a hyperplane that is used to separate two half spaces. This hyperplane is defined by a cutting dimension d and a cutting value v as $x_d = v$, where v denotes the d^{th} coordinate value.

#include <CGAL/Plane_separator.h>

Is Model for the Concepts

SpatialSeparator

Creation

Plane_separator<FT> s(int d, FT v);	Constructs a separator that separates two half spaces by a hyperplane defined by $x_d = v$, where v denotes the d^{th} coordinate value.
Plane_separator<FT> s(p);	Copy constructor.

Operations

void	*s.set_cutting_dimension(int d)*	
		Sets the cutting dimension to d.
void	*s.set_cutting_value(FT v)*	
		Sets the cutting value to v.
int	*s.cutting_dimension() const*	
		Returns the number of the cutting dimension.
FT	*s.cutting_value() const*	
		Returns the cutting value.
Plane_separator<FT>		
	$s = s2$	Assignment operator.

Class

Output Operators

template<class FT>
std::ostream& *std::ostream& os << s*

Inserts the plane separator *s* in the output stream *os* and returns *os*.

advanced

CGAL::Point_container<Traits>

advanced

Definition

A custom container for points used to build a tree. Each point container holds the points from a rectangle associated with a node of the tree. In the remainder of this reference page this rectangle is called the associated rectangle. Provides a method to split a container and a number of methods to support the implementation of splitting rules.

#include <CGAL/Point_container.h>

Parameters

Expects for the template argument an implementation for d-dimensional points of the concept *SearchTraits*, for example *CGAL::Cartesian_d<double>*.

Types

Traits::FT	*FT;*	Number type.
Traits::Point_d	*Point_d;*	Point type.
Point_container<Traits>:: iterator		An iterator with value type *Point_d**.
Point_container<Traits>:: const_iterator		A const iterator with value type *const Point_d**.

Creation

Point_container<Traits> c(int d);

Construct an empty container for storing d-dimensional points.

template <class InputIterator>
Point_container<Traits> c(int d, InputIterator begin, InputIterator end);

Construct the container of d-dimensional points of type *Point_d* given by the iterator sequence *[begin, end)*.

Operations

template <class SpatialSeparator>
void c.split(&c2, SpatialSeparator sep, bool sliding=false)

Given an empty container *c2* with the same dimension as *c*, splits *c* into *c*and *c2* using the separator *sep*. If sliding is *true* after splitting each container contains at least one point. Container *c* should contain at least two points.

void *c.swap(&c2)* Swap the contents of *c* and *c2*

void *c.recompute_tight_bounding_box()*

Recompute the bounding box of the points in the container.

iterator *c.begin()* Returns an iterator to a pointer to the first point.

iterator *c.end()* Returns the appropriate past-the-end iterator.

const_iterator *c.begin() const* Returns a const iterator to a pointer to the first point.

const_iterator *c.end() const* Returns the appropriate past-the-end const iterator.

int *c.dimension() const* Returns the dimension.

int *c.built_coordinate() const*

Returns coordinate for which the pointer list is built.

int *c.max_span_coord() const*

Returns coordinate where the associated rectangle has maximal span.

int *c.max_tight_span_coord() const*

Returns coordinate where the point coordinates have maximal span.

FT *c.max_span_lower() const*

Returns lower value of the interval corresponding to *max_span_coord()*.

FT *c.max_tight_span_lower() const*

Returns lower value of the interval corresponding to *max_tight_span_coord()*. That is, the smallest *max_tight_span_coord()*-th coordinate of the points in *c*.

FT *c.max_span_upper() const*

Returns upper value of the interval corresponding to *max_span_coord()*.

FT *c.max_span_upper_without_dim(int d) const*

Returns upper value of the interval over all dimensions without taking dimension *d* into account.

FT *c.max_tight_span_upper() const*

Returns upper value of the interval corresponding to *max_tight_span_coord()*.

FT *c.max_spread() const*

Returns the size of the interval corresponding to *max_span_coord()*.

FT *c.max_tight_spread() const*

Returns the size of the interval corresponding to *max_tight_span_coord()*.

FT *c.median(int split_coord) const*

Returns the median value of the points stored in the container for dimension *split_coord*.

Kd_tree_rectangle<Traits>

c.bounding_box() const

Returns the associated rectangle.

Kd_tree_rectangle<Traits>

c.tight_bounding_box()

Returns the bounding box of the items in associated rectangle.

int *c.max_tight_span_coord_balanced(FT aspect_ratio) const*

Returns the dimension with the maximal point spread, for which after fair splitting the ratio of the length of the longest side and the smallest side of the bounding box of the items in associated rectangle, does not exceed *aspect_ratio*.

FT *c.balanced_fair(int d, FT aspect_ratio)*

Returns the splitting value for fair splitting.

FT	*c.balanced_sliding_fair(int d, FT aspect_ratio)*	
		Returns the splitting value for sliding fair splitting.
std::size_t	*c.size() const*	Returns the number of points stored.
bool	*c.empty() const*	Returns true if no points are present, false otherwise.

Output Routines

template<class Traits> *std::ostream&*	*std::ostream& s << c*	Prints the point container *c* to the output stream *s* and returns *s*.

See Also

SearchTraits
SpatialSeparator

advanced

Concept

SearchTraits

Definition

The concept *SearchTraits* defines the requirements for the template parameter of the search classes.

Types

SearchTraits:: Point_d	Point type. *CGAL::Kernel_traits* has to be specialized for this type.
SearchTraits:: FT	The number type of the Cartesian coordinates of types *Point_d*
SearchTraits:: Cartesian_const_iterator_d	A random access iterator type to enumerate the Cartesian coordinates of a point.
SearchTraits:: Construct_cartesian_const_iterator_d	Functor with operators to construct iterators on the first and the past-the-end iterator for the Cartesian coordinates of a point. This functor must provides the type *result_type* that must be the same a *Cartesian_const_iterator_d*.

Creation

Operations

Construct_cartesian_const_iterator_d

t.construct_construct_cartesian_const_iterator_d_object(Point_d p) const

function used to construct an object of type *Construct_cartesian_const_iterator_d*.

Has Models

CGAL::Cartesian_d<FT>
CGAL::Homogeneous_d<RT>
CGAL::Search_traits_2<Kernel>
CGAL::Search_traits_3<Kernel>,
CGAL::Search_traits<NT,Point,CartesianCoordinateIterator,ConstructCartesianCoordinateIterator,ConstructMinVertex,ConstructM
.

See Also

RangeSearchTraits
CGAL::Search_traits_adapter<Key,PointPropertyMap,BaseTraits>

Concept

RangeSearchTraits

Definition

The concept *RangeSearchTraits* defines the requirements for the template parameter of the search classes. This concept also defines requirements to range search queries in a model of *SpatialTree*.

Refines

SearchTraits

Types

RangeSearchTraits:: Iso_box_d	Iso box type. It is only needed for range search queries.
RangeSearchTraits:: Sphere_d	Sphere type. It is only needed for range search queries.
RangeSearchTraits:: Construct_iso_box_d	Functor with operator to construct the iso box from two points.
RangeSearchTraits:: Construct_center_d	Functor with operator to construct the center of an object of type *Sphere_d*.
RangeSearchTraits:: Construct_squared_radius_d	Functor with operator to compute the squared radius of a an object of type *Sphere_d*.
RangeSearchTraits:: Construct_min_vertex_d	Functor with operator to construct the vertex with lexicographically smallest coordinates of an object of type *Iso_box_d*.
RangeSearchTraits:: Construct_max_vertex_d	Functor with operator to construct the vertex with lexicographically largest coordinates of an object of type *Iso_box_d*.

Has Models

CGAL::Cartesian_d<FT>
CGAL::Homogeneous_d<RT>
CGAL::Search_traits_2<Kernel>
CGAL::Search_traits_3<Kernel>

See Also

SearchTraits
CGAL::Search_traits_adapter<Key,PointPropertyMap,BaseTraits>

CGAL::Search_traits_2<Kernel>

Definition

The class *Search_traits_2<Kernel>* can be used as a template parameter of the kd tree and the search classes.

#include <CGAL/Search_traits_2.h>

Parameters

Expects for the template argument a model of the concept *Kernel*, for example *CGAL::Cartesian<double>* or *CGAL::Simple_cartesian<CGAL::Gmpq>*.

Is Model for the Concepts

SearchTraits and *RangeSearchTraits*.

Types

Kernel::FT	*FT;*	Number type.
Kernel::Point_2	*Point_d;*	Point type.
Kernel::Iso_rectangle_2	*Iso_box_d;*	Iso box type.
Kernel::Sphere_2	*Sphere_d;*	Sphere type.
Kernel::Cartesian_const_iterator_2	*Cartesian_const_iterator_d;*	An iterator over the Cartesian coordinates.
Kernel::Construct_cartesian_const_iterator_2	*Construct_cartesian_const_iterator_d;*	A functor with two function operators, which return the begin and past the end iterator for the Cartesian coordinates. The functor for begin has as argument a *Point_d*. The functor for the past the end iterator, has as argument a *Point_d* and an *int*.

Kernel::Construct_iso_rectangle_2

Construct_iso_box_d;	Functor with operator to construct the iso box from two points.

Kernel::Construct_center_2

Construct_center_d;	Functor with operator to construct the center of an object of type *Sphere_d*.

Kernel::Compute_squared_radius_2

Construct_squared_radius_d;

Functor with operator to compute the squared radius of a an object of type *Sphere_d*.

Kernel::Construct_min_vertex_2

Construct_min_vertex_d;

Functor with operator to construct the vertex with lexicographically smallest coordinates of an object of type *Iso_box_d*.

Kernel::Construct_max_vertex_2

Construct_max_vertex_d;

Functor with operator to construct the vertex with lexicographically largest coordinates of an object of type *Iso_box_d*.

See Also

Search_traits_3<Kernel>
Search_traits<NT_,Point,CartesianConstIterator,ConstructCartesianConstIterator

CGAL::Search_traits_3<Kernel>

Class

Definition

The class *Search_traits_3<Kernel>* can be used as a template parameter of the kd tree and the search classes. *Kernel* must be a CGAL kernel.

#include <CGAL/Search_traits_3.h>

Parameters

Expects for the template argument a model of the concept *Kernel*, for example *CGAL::Cartesian<double>* or *CGAL::Simple_cartesian<CGAL::Gmpq>*.

Is Model for the Concepts

SearchTraits and *RangeSearchTraits*.

Types

Kernel::FT	*FT;*	Number type.
Kernel::Point_3	*Point_d;*	Point type.
Kernel::Iso_cuboid_3		
	Iso_box_d;	Iso box type.
Kernel::Sphere_3	*Sphere_d;*	Sphere type.
Kernel::Cartesian_const_iterator_3		
	Cartesian_const_iterator_d;	
		An iterator over the Cartesian coordinates.
Kernel::Construct_cartesian_const_iterator_3		
	Construct_cartesian_const_iterator_d;	
		A functor with two function operators, which return the begin and past the end iterator for the Cartesian coordinates. The functor for begin has as argument a *Point_d*. The functor for the past the end iterator, has as argument a *Point_d* and an *int*.

Kernel::Construct_center_3

Construct_center_d; Functor with operator to construct the center of an object of type *Sphere_d*.

Kernel::Compute_squared_radius_3

Construct_squared_radius_d;

Functor with operator to compute the squared radius of a an object of type *Sphere_d*.

Kernel::Construct_min_vertex_3

Construct_min_vertex_d;

Functor with operator to construct the vertex with lexicographically smallest coordinates of an object of type *Iso_box_d*.

Kernel::Construct_max_vertex_3

Construct_max_vertex_d;

Functor with operator to construct the vertex with lexicographically largest coordinates of an object of type *Iso_box_d*.

See Also

Search_traits_2<Kernel>
Search_traits<Point,CartesianConstIterator,ConstructCartesianConstIterator

CGAL::Search_traits_d<Kernel>

Class

Definition

The class *Search_traits_d<Kernel>* can be used as a template parameter of the kd tree and the search classes. *Kernel* must be a CGAL kernel.

Kernel must be a d-dimensional CGAL kernel.

#include <CGAL/Search_traits_d.h>

Parameters

Expects for the template argument a model of the concept *Kernel_d*, for example *CGAL::Cartesian_d<double>* or *CGAL::Homogeneous_d<CGAL::Gmpz>*.

Is Model for the Concepts

SearchTraits and *RangeSearchTraits*.

Types

Kernel::FT	*NT;*	Number type.
Kernel::Point_d	*Point_d;*	Point type.
Kernel::Iso_box_d	*Iso_box_d;*	Iso box type.
Kernel::Sphere_d	*Sphere_d;*	Sphere type.

Kernel::Cartesian_const_iterator_d

Cartesian_const_iterator;

An iterator over the Cartesian coordinates.

Kernel::Construct_cartesian_const_iterator_d

Construct_cartesian_const_iterator;

A functor with two function operators, which return the begin and past the end iterator for the Cartesian coordinates. The functor for begin has as argument a *Point_d*. The functor for the past the end iterator, has as argument a *Point_d* and an *int*.

Kernel::Construct_min_vertex_d

Construct_min_vertex_d;

Functor with operator to construct the vertex with lexicographically smallest coordinates of an object of type *Iso_box_d*.

Kernel::Construct_max_vertex_d

Construct_max_vertex_d;

Functor with operator to construct the vertex with lexicographically largest coordinates of an object of type *Iso_box_d*.

See Also

Search_traits_2<Kernel>
Search_traits_3<Kernel>
Search_traits<Point,CartesianConstIterator,ConstructCartesianConstIterator>

CGAL::Search_traits<NT,Point,CartesianIterator,ConstructCartesianIterator,Constru

Definition

The class *Search_traits<NT,Point,CartesianIterator,ConstructCartesianIterator,ConstructMinVertex,ConstructMaxVertex>* can be used as a template parameter of the kd tree and the search classes. It is a mere wrapper for the geometric types needed by these classes.

#include <CGAL/Search_traits.h>

Is Model for the Concepts

SearchTraits.

Types

NT	*FT;*	The number type of the coordinates.
Point	*Point_d;*	Point type.
CartesianIterator	*Cartesian_const_iterator_d;*	An iterator over the coordinates.
ConstructCartesianIterator	*Construct_Cartesian_const_iterator_d;*	A functor with two function operators, which return the begin and past the end iterator for the Cartesian coordinates. The functor for begin has as argument a *Point_d*. The functor for the past the end iterator, has as argument a *Point_d* and an *int*.
ConstructMinVertex	*Construct_min_vertex_d;*	Functor with operator to construct the vertex with lexicographically smallest coordinates of an object of type *Iso_box_d*.
Kernel::ConstructMaxVertex	*Construct_max_vertex_d;*	Functor with operator to construct the vertex with lexicographically largest coordinates of an object of type *Iso_box_d*.

See Also

Search_traits_2<Kernel>
Search_traits_3<Kernel>
Search_traits_d<Kernel>

CGAL::Search_traits_adapter<Key,PointPropertyMap,BaseTraits>

Class

Definition

The class *Search_traits_adapter<Key,PointPropertyMap,BaseTraits>* can be used as a template parameter of the kd tree and the search classes. When using this traits class in a nearest neighbor search algorithm, the class *Distance_for_point_adapter<Key,PointPropertyMap,Base_distance>* must be used as distance.

#include <CGAL/Search_traits_adapter.h>

Parameters

Key is a type that is associated to a point of type *Base_distance::Point_d*.
PointPropertyMap is a model of boost::ReadablePropertyMap with *Key* as *key_type* and *Base_distance::Point_d* as *value_type*.
BaseTraits is a model of either *SearchTraits* or *RangeSearchTraits*.

Inherits From

BaseTraits

Is Model for the Concepts

SearchTraits if *BaseTraits* is a model of *SearchTraits*.
RangeSearchTraits if *BaseTraits* is a model of *RangeSearchTraits*.

Types

Key	*Point_d;*
BaseTraits::FT	*FT;*
BaseTraits::Cartesian_const_iterator_d	*Cartesian_const_iterator_d;*
BaseTraits	*Base;*

Creation

Search_traits_adapter<Key,PointPropertyMap,BaseTraits> t(PointPropertyMap ppmap=PointPropertyMap(), BaseTraits base=BaseTraits())

Constructor initializing the class to *base* and setting the point property map of the class to *ppmap*.

Operations

PointPropertyMap *t.point_property_map() const*

Returns the point property map.

See Also

Distance_for_point_adapter<Key,PointPropertyMap,Base_distance>
Search_traits_2<Kernel>
Search_traits_3<Kernel>
Search_traits_d<Kernel>
Search_traits<Point,CartesianConstIterator,ConstructCartesianConstIterator

CGAL::Distance_for_point_adapter<Key,PointPropertyMap,Base_distance>

Class

Definition

A class that uses a point property map to adapt a distance class to work on a key as point type. When using *Search_traits_adapter<Key,PointPropertyMap,BaseTraits>* in a nearest neighbor search algorithm, this class must be used as distance.

#include <CGAL/Search_traits_adapter.h>

Parameters

Key is a type that is associated to a point of type *Base_distance::Point_d*.
PointPropertyMap is a model of boost::ReadablePropertyMap with *Key* as *key_type* and *Base_distance::Point_d* as *value_type*.
Base_distance is a model of either *GeneralDistance* or *OrthogonalDistance*.

Inherits From

Base_distance

Is Model for the Concepts

GeneralDistance if *Base_distance* is a model of *GeneralDistance*.
OrthogonalDistance if *Base_distance* is a model of *OrthogonalDistance*.

Types

Base_distance::FT	*FT;*
Key	*Point_d;*
Base_distance::Query_item	*Query_item;*

Creation

Distance_for_point_adapter<Key,PointPropertyMap,Base_distance> d(PointPropertyMap ppmap=PointPropertyMap(), Base_distance base=Base_distance())

Constructor initializing the class to *base* and setting the point property map of the class to *ppmap*.

Operations

PointPropertyMap   *d.point_property_map() const*

Returns the point property map.

See Also

Search_traits_adapter<Key,PointPropertyMap,BaseTraits>

CGAL::Sliding_fair<Traits, SpatialSeparator>

Functor
Class

Definition

Implements the *sliding fair* splitting rule. This splitting rule is a compromise between the *Fair* splitting rule and the *Sliding_midpoint* rule. Sliding fair-split is based on the theory that there are two types of splits that are good: balanced splits that produce fat rectangles, and unbalanced splits provided the rectangle with fewer points is fat.

Also, this splitting rule maintains an upper bound on the maximal allowed ratio of the longest and shortest side of a rectangle (the value of this upper bound is set in the constructor of the fair splitting rule). Among the splits that satisfy this bound, it selects the one one in which the points have the largest spread. It then considers the most extreme cuts that would be allowed by the aspect ratio bound. This is done by dividing the longest side of the rectangle by the aspect ratio bound. If the median cut lies between these extreme cuts, then we use the median cut. If not, then consider the extreme cut that is closer to the median. If all the points lie to one side of this cut, then we slide the cut until it hits the first point. This may violate the aspect ratio bound, but will never generate empty cells.

#include <CGAL/Splitters.h>

Parameters

Expects for the first template argument a model of the concept *SearchTraits*, for example *CGAL::Cartesian_d< double>*.

Expects for the second template argument a model of the concept *Separator*. It has as default value the type, *CGAL::Plane_separator<Traits::FT>*

Is Model for the Concepts

Splitter

Types

Traits::FT	*FT;*	Number type.

Creation

Sliding_fair<Traits, SpatialSeparator> s(unsigned int bucket_size, FT aspect_ratio=FT(3));

Constructor.

Operations

FT	*s.aspect_ratio()*	Returns the maximal ratio between the largest and smallest side of a cell allowed for fair splitting.
unsigned int	*s.bucket_size()*	Returns the bucket size of the leaf nodes.

See Also

Splitter
SpatialSeparator

CGAL::Sliding_midpoint<Traits, SpatialSeparator>

Functor
Class

Definition

Implements the *sliding midpoint* splitting rule. This is a modification of the *Midpoint_of_rectangle* splitting rule. It first attempts to perform a midpoint of rectangle split as described above. If data points lie on both sides of the separating plane the sliding midpoint rule computes the same separator as the midpoint of rectangle rule. If the data points lie only on one side it avoids this by sliding the separator, computed by the midpoint of rectangle rule, to the nearest data point.

#include <CGAL/Splitters.h>

Parameters

Expects for the first template argument a model of the concept *SearchTraits*, for example *CGAL::Cartesian_d< double>*.

Expects for the second template argument a model of the concept *Separator*. It has as default value the type, *CGAL::Plane_separator<Traits::FT>*.

Is Model for the Concepts

Splitter

Creation

Sliding_midpoint<Traits, SpatialSeparator> s(unsigned int bucket_size);

Constructor.

Operations

unsigned int	*s.bucket_size()*	Returns the bucket size of the leaf nodes.

See Also

Splitter
SpatialSeparator

Concept

SpatialSeparator

advanced

Definition

The concept SpatialSeparator defines the requirements for a separator. A separator is a (d-1)-dimensional subspace that separates a d-dimensional space into two parts. One part of space is said to be on the negative side of the separator and the other part of space is said to be on the positive side of the separator.

Has Models

CGAL::Plane_separator<FT>.

Types

SpatialSeparator:: FT	Number type.

Creation

SpatialSeparator s;	Default constructor.

Operations

void	*s.set_cutting_dimension(int d)*	
		Sets the cutting dimension to d.
void	*s.set_cutting_value(FT v)*	
		Sets the cutting value to v.
int	*s.cutting_dimension()*	Returns the number of the cutting dimension.
FT	*s.cutting_value()*	Returns the cutting value.
template <class Point_d> *bool*	*s.has_on_negative_side(Point_d p)*	
		Returns true if and only if the point p is on the negative side of the separator.

advanced

SpatialTree

Definition

The concept SpatialTree defines the requirements for a tree supporting both neighbor searching and approximate range searching.

Types

SpatialTree:: SearchTraits	Search traits.
SpatialTree:: Point_d	Point type.
SpatialTree:: iterator	Bidirectional const iterator with value type *Point_d* that allows to enumerate all points in the tree.
SpatialTree:: Node_handle	Node handle.
SpatialTree:: Node_const_handle	const node handle.
SpatialTree:: Point_d_iterator	const iterator with value type *const Point_d**.
SpatialTree:: Splitter	Splitter.
SpatialTree:: Distance	Distance.

Creation

template <class InputIterator>
SpatialTree tree(InputIterator first, InputIterator beyond, SearchTraits t);

Constructs a tree on the elements from the sequence *[first,beyond)*.

Operations

template <class OutputIterator, class FuzzyQueryItem>
OutputIterator *tree.search(OutputIterator it, FuzzyQueryItem q)*

Reports the points that are approximately contained by q. The value type of *OutputIterator* must be *Point_d*.

iterator *tree.begin() const* Returns a const iterator to the first point in the tree.

iterator	*tree.end() const*	Returns the appropriate past-the-end const iterator.
Node_handle	*tree.root()*	Returns a handle to the root node of the tree.
Node_const_handle	*tree.root() const*	Returns a const handle to the root node of the tree.
Kd_tree_rectangle<SearchTraits::FT>	*tree.bounding_box() const*	returns a const reference to the bounding box of the root node of the tree.
unsigned int	*tree.size() const*	Returns the number of items that are stored in the tree.

Has Models

CGAL::Kd_tree<Traits,Splitter,UseExtendedNode>.

Splitter

advanced

Definition

The concept Splitter defines the requirements for a function object class implementing a splitting rule.

Has Models

CGAL::Fair<Traits, SpatialSeparator>,
CGAL::Median_of_rectangle<Traits, SpatialSeparator>,
CGAL::Median_of_max_spread<Traits, SpatialSeparator>,
CGAL::Midpoint_of_rectangle<Traits, SpatialSeparator>,
CGAL::Midpoint_of_max_spread<Traits, SpatialSeparator>,
CGAL::Sliding_fair<Traits, SpatialSeparator>,
CGAL::Sliding_midpoint<Traits, SpatialSeparator>.

Types

Splitter:: FT	Number type.
Splitter:: Separator	Separator.
Splitter:: Container	Typedef to an instantiation of *CGAL::Point_container< Traits>*.

The parameters *aspect_ratio* and *bucket_size* define the way in which *k-d* tree is constructed.

Operations

FT	*s.aspect_ratio() const*	Returns the maximal ratio between the largest and smallest side of a cell allowed for fair splitting.
unsigned int	*s.bucket_size() const*	Returns the bucket size of the leaf nodes.
void	*s(Separator& sep, Container& c0, Container& c1)*	

Sets up *sep* and splits points of *c0* into *c0* and *c1* using *sep*. Container *c0* should contain at least two points and *c1* must be empty.

advanced

Class

CGAL::Weighted_Minkowski_distance<Traits>

Definition

The class *Weighted_Minkowski_distance<Traits>* provides an implementation of the concept *OrthogonalDistance*, with a weighted Minkowski metric on d-dimensional points defined by $l_p(w)(r,q) = (\Sigma_{i=1}^{i=d} w_i(r_i - q_i)^p)^{1/p}$ for $0 < p < \infty$ and defined by $l_\infty(w)(r,q) = max\{w_i|r_i - q_i| \mid 1 \leq i \leq d\}$. For the purpose of the distance computations it is more efficient to compute the transformed distance $\sigma_{i=1}^{i=d} w_i(r_i - q_i)^p$ instead of the actual distance.

#include <CGAL/Weighted_Minkowski_distance.h>

Parameters

Expects for the template argument a model of the concept *SearchTraits*, for example *CGAL::Search_traits_2< Kernel>*.

Is Model for the Concepts

OrthogonalDistance

Types

Traits::FT	*FT;*	Number type.
Traits::Point_d	*Point_d;*	Point type.

Creation

Weighted_Minkowski_distance<Traits> wd(int d, Traits t=Traits());

Constructor implementing l_2 metric for d-dimensional points.

template <class InputIterator>
Weighted_Minkowski_distance<Traits> wd(FT power,
int dim,
InputIterator wb,
InputIterator we,
Traits t=Traits())

Constructor implementing the $l_{power}(weights)$ metric. $power \leq 0$ denotes the $l_\infty(weights)$ metric. The values in the iterator range *[wb,we)* are the weight.

Operations

FT *wd.transformed_distance(Point_d q, Point_d r) const*

Returns d^{power}, where d denotes the distance between q and r.

FT *wd.min_distance_to_rectangle(Point_d q, Kd_tree_rectangle<FT> r;) const*

Returns d^{power}, where d denotes the distance between the query item q and the point on the boundary of r closest to q.

FT *wd.max_distance_to_rectangle(Point_d q, Kd_tree_rectangle<FT> r;) const*

Returns d^{power}, where d denotes the distance between the query item q and the point on the boundary of r farthest to q.

FT *wd.new_distance(FT dist, FT old_off, FT new_off, int cutting_dimension) const*

Updates *dist* incrementally and returns the updated distance.

FT *wd.transformed_distance(FT d) const*

Returns d^p for $0 < p < \infty$. Returns d for $p = \infty$.

FT *wd.inverse_of_transformed_distance(FT d) const*

Returns $d^{1/p}$ for $0 < p < \infty$. Returns d for $p = \infty$.

See Also

OrthogonalDistance
CGAL::Euclidean_distance<Traits>

Chapter 62

dD Range and Segment Trees

Gabriele Neyer

Contents

62.1 Introduction

This chapter presents the CGAL range tree and segment tree data structures.

62.2 Definitions

This section presents d-dimensional range and segment trees. A one-dimensional range tree is a binary search tree on *one-dimensional point data*. Here we call all one-dimensional data types having a strict ordering (like integer and double) *point data*. *d-dimensional point data* are d-tuples of one-dimensional point data.

A one-dimensional segment tree is a binary search tree as well, but with *one-dimensional interval data* as input data. One-dimensional interval data is a pair (i.e., 2-tuple) (a,b), where a and b are one-dimensional point data of the same type and $a < b$. The pair (a,b) represents a half open interval $[a,b)$. Analogously, a d-dimensional interval is represented by a d-tuple of one-dimensional intervals.

The *input data type* for a d-dimensional tree is a container class consisting of a d-dimensional point data type, interval data type or a mixture of both, and optionally a *value type*, which can be used to store arbitrary data. E.g., the d–dimensional bounding box of a d–dimensional polygon may define the interval data of a d–dimensional segment tree and the polygon itself can be stored as its value. An *input data item* is an instance of an input data type.

The range and segment tree classes are fully generic in the sense that they can be used to define *multilayer trees*. A multilayer tree of dimension (number of layers) d is a simple tree in the d-th layer, whereas the k-th layer, $1 \leq k \leq d-1$, of the tree defines a tree where each (inner) vertex contains a multilayer tree of dimension $d-k+1$. The $k-1$-dimensional tree which is nested in the k-dimensional tree (T) is called the *sublayer tree* (of T). For example, a d-dim tree can be a range tree on the first layer, constructed with respect to the first dimension of d-dimensional data items. On all the data items in each subtree, a $(d-1)$-dimensional tree is built, either a range or a segment tree, with respect to the second dimension of the data items. And so on. Figures 62.2, 62.3 and 62.5 illustrate the meaning of a sublayer tree graphically.

After creation of the tree, further insertions or deletions of data items are disallowed. The tree class does neither depend on the type of data nor on the concrete physical representation of the data items. E.g., let a multilayer tree be a segment tree for which each vertex defines a range tree. We can choose the data items to consist of intervals of type *double* and the point data of type *integer*. As value type we can choose *string*.

For this generality we have to define what the tree of each dimension looks like and how the input data is organized. For dimension k, $1 \leq k \leq 4$, CGAL provides ready-to-use range and segment trees that can store k-dimensional keys (intervals resp.). Examples illustrating the use of these classes are given in Sections 62.5.1 and 62.6.1. The description of the functionality of these classes as well as the definition of higher dimensional trees and mixed multilayer trees is given in the reference manual.

In the following two sections we give short definitions of the version of the range tree and segment tree implemented here together with some examples. The presentation closely follows [dBvKOS97].

62.3 Software Design

In order to be able to define a multilayer tree we first designed the range and segment tree to have a template argument defining the type of the sublayer tree. With this sublayer tree type information the sublayers could be created. This approach lead to nested template arguments, since the sublayer tree can again have a template argument defining the sublayer. Therefore, the internal class and function identifiers got longer than a compiler-dependent limit. This happend already for $d = 2$.

Therefore, we chose another, object oriented, design. We defined a pure virtual base class called *Tree_base* from which we derived the classes *Range_tree_d* and *Segment_tree_d*. The constructor of these classes expects an argument called *sublayer_prototype* of type *Tree_base*. Since class *Range_tree_d* and class *Segment_tree_d* are derived from class *Tree_base*, one can use an instantiation of class *Range_tree_d* or class *Segment_tree_d* as constructor argument. This argument defines the sublayer tree of the tree. E.g., you can construct a *Range_tree_d* with an instantiation of class *Segment_tree_d* as constructor argument. You then have defined a range tree with a segment tree as sublayer tree. Since both classes *Range_tree_d* and *Segment_tree_d* expect a sublayer tree in their constructor we had to derive a third class called *Tree_anchor* from class *Tree_base* which does not expect a constructor argument. An instantiation of this class is used as constructor argument of class *Range_tree_d* or *Segment_tree_d* in order to stop the recursion.

All classes provide a *clone()* function which returns an instance (a copy) of the same tree type. The *clone()* function of the *sublayer_prototype* is called in the construction of the tree. In case that the sublayer tree again has a sublayer, it also has a *sublayer_prototype* which is also cloned and so on. Thus, a call to the *clone()* function generates a sublayer tree which has the complete knowledge about its sublayer tree.

The trees allow to perform window queries, enclosing queries, and inverse range queries on the keys. Clearly, an inverse range query makes only sense in the segment tree. In order to perform an inverse range query, a range query of ε width has to be performed. We prefered not to offer an extra function for this sort of query, since the inverse range query is a special case of the range query. Furthermore, offering an inverse range query in the segment tree class implies offering this function also in the range tree class and having an extra item in the traits class that accesses the inverse range query point.

The trees are templatized with three arguments: *Data, Window* and *Traits*. Type *Data* defines the input data type and type *Window* defines the query window type. The tree uses a well defined set of functions in order to access data. These functions have to be provided by class *Traits*.

The design partly follows the *prototype design pattern* in [GHJV95]. In comparison to our first approach using templates we want to note the following: In this approach the sublayer type is defined in use of object oriented programming at run time, while in the approach using templates, the sublayer type is defined at compile time.

The runtime overhead caused in use of virtual member functions in this object oriented design is negligible since all virtual functions are non trivial. The design concept is illustrated in Figure 62.1.

E.g. in order to define a two dimensional multilayer tree, which consists of a range tree in the first dimension and a segment tree in the second dimension we proceed as follows: We construct an object of type *Tree_anchor* which stops the recursion. Then we construct an object of type *Segment_tree_d*, which gets as prototype argument our object of type *Tree_anchor*. After that, we define an object of type *Range_tree_d* which is constructed with the object of type *Segment_tree_d* as prototype argument. The following piece of code illustrates the construction of the two-dimensional multilayer tree.

```
int main(){
  Tree_Anchor *anchor=new Tree_Anchor;
  Segment_Tree_d *segment_tree = new Segment_Tree_d(*anchor);
  Range_Tree_d *range_segment_tree = new Range_Tree_d(*segment_tree);
  /* let data_items be a list of Data items */
  range_segment_tree->make_tree(data_items.begin(),data_items.end());
}
```

Here, class *Tree_Anchor, Segment_Tree_d*, and *Range_Tree_d* are defined by *typedef*s:

```
typedef Tree_anchor<Data,Window> Tree_Anchor;
typedef Segment_tree_d<Data,Window,Interval_traits> Segment_Tree_d;
typedef Range_tree_d<Data,Window,Point_traits> Range_Tree_d;
```

Class *Tree_base* and class *Tree_anchor* get two template arguments: a class *Data* which defines the type of data that is stored in the tree, and a class *Window* which defines the type of a query range. The derived classes *Range_tree_d* and *Segment_tree_d* additionally get an argument called *Tree_traits* which defines the interface between the *Data* and the tree. Let the *Data* type be a d-dimensional tuple, which is either a point data or an interval data in each dimension. Then, the class *Tree_traits* provides accessors to the point (resp. interval) data of that tree layer and a compare function. Remind our example of the two-dimensional tree which is a range tree in the first dimension and a segment tree in the second dimension. Then, the *Tree_traits* class template argument of class *Segment_tree_d* defines an accessor to the interval data of the *Data*, and the *Tree_traits* class template argument of class *Range_tree_d* defines an accessor to the point data of *Data*. An example implementation for these classes is listed below.

```
struct Data{
```

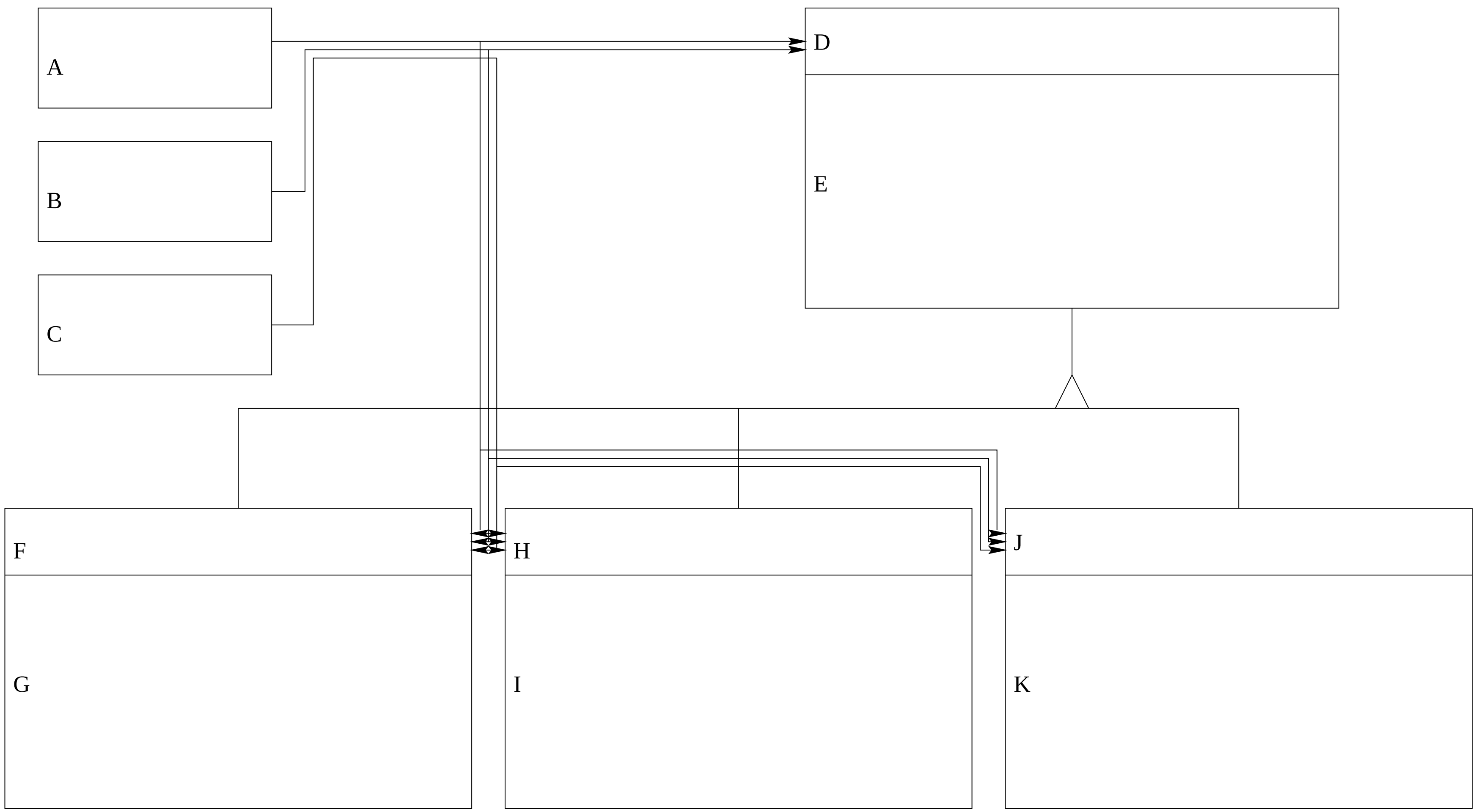


Figure 62.1: Design of the range and segment tree data structure. The symbol triangle means that the lower class is derived from the upper class.

```
  int min,max;  /* interval data */
  double point; /* point data    */
};

struct Window{
  int min,max;
  double min_point, max_point;
};

class Point_traits{
 public:
  typedef double Key;
  Key get_key(Data&  d){return d.point;} /*key accessor */
  Key get_left(Window& w){return w.min_point;}
  Key get_right(Window& w){return w.max_point;}
  bool comp(Key& key1, Key& key2){return (key1 < key2);}
}

class Interval_traits{
 public:
  typedef int Key;
  Key get_left(Data&  d){return d.min;}
  Key get_right(Data&  d){return d.max;}
  Key get_left_win(Window& w){return w.min;}
  Key get_right_win(Window& w){return w.max;}
  static bool comp(Key& key1, Key& key2){return (key1 < key2);}
}
```

62.4 Creating an Arbitrary Multilayer Tree

Now let us have a closer look on how a multilayer tree is built. In case of creating a d-dimensional tree, we handle a sequence of arbitrary data items, where each item defines a d-dimensional interval, point or other object. The tree is constructed with an iterator over this structure. In the i-th layer, the tree is built with respect to the data slot that defines the i-th dimension. Therefore, we need to define which data slot corresponds to which dimension. In addition we want our tree to work with arbitrary data items. This requires an adaptor between the algorithm and the data item. This is resolved by the use of traits classes, implemented in form of a traits class using function objects. These classes provide access functions to a specified data slot of a data item. A d-dimensional tree is then defined separately for each layer by defining a traits class for each layer.

62.5 Range Trees

A one-dimensional range tree is a binary search tree on one-dimensional point data. The point data of the tree is stored in the leaves. Each inner vertex stores the highest entry of its left subtree. The version of a range tree implemented here is static, which means that after construction of the tree, no elements be inserted or deleted. A d-dimensional range tree is a binary leaf search tree according to the first dimension of the d-dimensional point data, where each vertex contains a $(d-1)$-dimensional search tree of the points in the subtree (sublayer tree) with respect to the second dimension. See [dBvKOS97] and [Sam90] for more detailed information.

A d-dimensional range tree can be used to determine all d-dimensional points that lie inside a given d-dimensional interval (*window_query*). Figure 62.2 shows a two-dimensional range tree, Figure 62.3 a d-

dimensional one.

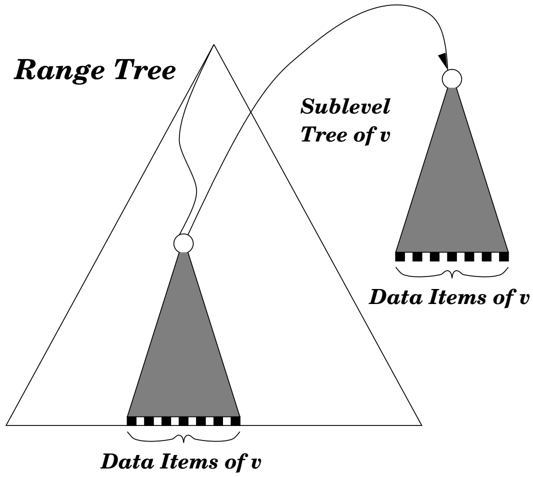


Figure 62.2: A two-dimensional range tree. The tree is a binary search tree on the first dimension. Each sublayer tree of a vertex *v* is a binary search tree on the second dimension. The data items in a sublayer tree of *v* are all data items of the subtree of *v*.

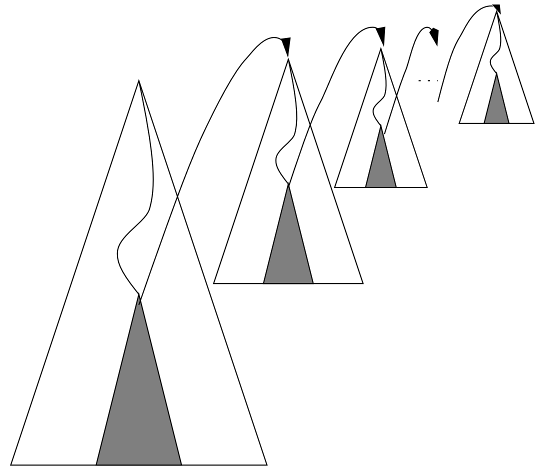


Figure 62.3: A d-dimensional range tree. For each layer of the tree, one sublayer tree is illustrated.

The tree can be built in $O(n \log^{d-1} n)$ time and needs $O(n \log^{d-1} n)$ space. The d-dimensional points that lie in the d-dimensional query interval can be reported in $O(\log^d n + k)$ time, where n is the total number of points and k is the number of reported points.

62.5.1 Example for Range Tree on Map-like Data

The following example program uses the predefined *Range_tree_2* data structure together with the predefined traits class *Range_tree_map_traits_2* which has two template arguments specifying the type of the point data in each dimension (*CGAL::Cartesian*<*double*>) and the value type of the 2-dimensional point data (*char*). Therefore the *Range_tree_2* is defined on 2-dimensional point data each of which is associated with a character. Then, a few data items are created and put into a list. After that the tree is constructed according to that list, a window query is performed, and the query elements are given out.

```
#include <CGAL/Cartesian.h>
#include <CGAL/Range_segment_tree_traits.h>
#include <CGAL/Range_tree_k.h>

typedef CGAL::Cartesian<double> K;
typedef CGAL::Range_tree_map_traits_2<K, char> Traits;
typedef CGAL::Range_tree_2<Traits> Range_tree_2_type;

int main()
{
  typedef Traits::Key Key;
  typedef Traits::Interval Interval;
```

```
  std::vector<Key> InputList, OutputList;
  InputList.push_back(Key(K::Point_2(8,5.1), 'a'));
  InputList.push_back(Key(K::Point_2(1,1.1), 'b'));
  InputList.push_back(Key(K::Point_2(3,2.1), 'c'));

  Range_tree_2_type Range_tree_2(InputList.begin(),InputList.end());
  Interval win(Interval(K::Point_2(4,8.1),K::Point_2(5,8.2)));
  std::cout << "\n Window Query:\n ";
  Range_tree_2.window_query(win, std::back_inserter(OutputList));
  std::vector<Key>::iterator current=OutputList.begin();
  while(current!=OutputList.end()){
    std::cout << (*current).first.x() << "," << (*current).first.y()
         << ":" << (*current++).second << std::endl;
  }
}
```

62.5.2 Example for Range Tree on Set-like Data

This example illustrates the use of the range tree on 2-dimensional point data (no value is associated to a data item). After the definition of the tree, some input data items are created and the tree is constructed according to the input data items. After that, a window query is performed and the query elements are given to standard out.

```
#include <CGAL/Cartesian.h>
#include <CGAL/Range_segment_tree_traits.h>
#include <CGAL/Range_tree_k.h>

typedef CGAL::Cartesian<double> K;
typedef CGAL::Range_segment_tree_set_traits_2<K> Traits;
typedef CGAL::Range_tree_2<Traits> Range_tree_2_type;

int main()
{
  typedef Traits::Key Key;
  typedef Traits::Interval Interval;
  std::vector<Key> InputList, OutputList;
  std::vector<Key>::iterator first, last, current;

  InputList.push_back(Key(8,5.1));
  InputList.push_back(Key(1,1.1));
  InputList.push_back(Key(3,2.1));

  Range_tree_2_type Range_tree_2(InputList.begin(),InputList.end());

  Interval win=Interval(Key(4,8.1),Key(5,8.2));
  std::cout << std::endl << "Window Query: lower left point: (4.0,5.0),";
  std::cout << "upper right point: (8.1,8.2)" << std::endl;
  Range_tree_2.window_query(win, std::back_inserter(OutputList));
  current=OutputList.begin();
  while(current!=OutputList.end()){
    std::cout << (*current).x()<< "-" << (*current).y() << std::endl;
    current++;
```

```
  }
}
```

62.6 Segment Trees

A segment tree is a static binary search tree for a given set of coordinates. The set of coordinates is defined by the endpoints of the input data intervals. Any two adjacent coordinates build an elementary interval. Every leaf corresponds to an elementary interval. Inner vertices correspond to the union of the subtree intervals of the vertex. Each vertex or leaf v contains a sublayer type (or a list, if it is one-dimensional) that will contain all intervals I, such that I contains the interval of vertex v but not the interval of the parent vertex of v.

A d-dimensional segment tree can be used to solve the following problems:

- Determine all d-dimensional intervals that contain a d-dimensional point. This query type is called "inverse range query".
- Determine all d-dimensional intervals that enclose a given d-dimensional interval (*enclosing_query*).
- Determine all d-dimensional intervals that partially overlap or are contained in a given d-dimensional interval (*window_query*).

In Figure 62.4 an example of a one-dimensional segment tree is given. Figure 62.5 shows a two-dimensional segment tree.

The tree can be built in $O(n \log^d n)$ time and needs $O(n \log^d n)$ space. The processing time for inverse range queries in an d-dimensional segment tree is $O(\log^d n + k)$ time, where n is the total number of intervals and k is the number of reported intervals.

One possible application of a two-dimensional segment tree is the following. Given a set of convex polygons in two-dimensional space (CGAL::Polygon_2), we want to determine all polygons that intersect a given rectangular query window. Therefore, we define a two-dimensional segment tree, where the two-dimensional interval of a data item corresponds to the bounding box of a polygon and the value type corresponds to the polygon itself. The segment tree is created with a sequence of all data items, and a window query is performed. The polygons of the resulting data items are finally tested independently for intersections.

62.6.1 Example for Segment Tree on Map-like Data

The following example program uses the predefined *Segment_tree_2* data structure together with the predefined traits class *Segment_tree_map_traits_2* which has two template arguments specifying the type of the point data in each dimension (*CGAL::Cartesian<double>*) and the value type of the 2-dimensional point data (*char*). Therefore the *Segment_tree_2* is defined on 2-dimensional point data (*CGAL::Point_2<Cartesian<double> >*) each of which is associated with a character. Then, a few data items are created and put into a list. After that the tree is constructed according to that list, a window query is performed, and the query elements are given out.

```
#include <CGAL/Cartesian.h>
#include <CGAL/Segment_tree_k.h>
#include <CGAL/Range_segment_tree_traits.h>

typedef CGAL::Cartesian<double> K;
```

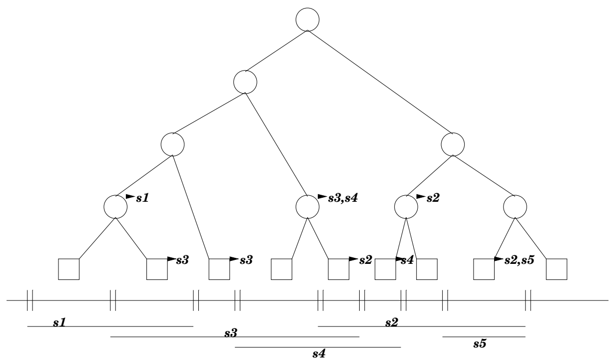


Figure 62.4: A one-dimensional segment tree. The segments and the corresponding elementary intervals are shown below the tree. The arcs from the nodes point to their subsets.

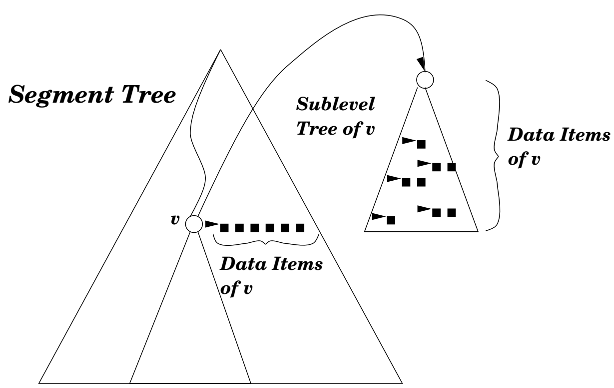


Figure 62.5: A two-dimensional segment tree. The first layer of the tree is built according to the elementary intervals of the first dimension. Each sublayer tree of a vertex v is a segment tree according to the second dimension of all data items of v.

```
typedef CGAL::Segment_tree_map_traits_2<K, char> Traits;
typedef CGAL::Segment_tree_2<Traits > Segment_tree_2_type;

int main()
{
  typedef Traits::Interval Interval;
  typedef Traits::Pure_interval Pure_interval;
  typedef Traits::Key Key;
  std::list<Interval> InputList, OutputList1, OutputList2;

  InputList.push_back(Interval(Pure_interval(Key(1,5), Key(2,7)),'a'));
  InputList.push_back(Interval(Pure_interval(Key(2,7), Key(3,8)),'b'));
  InputList.push_back(Interval(Pure_interval(Key(6,9), Key(9,13)),'c'));
  InputList.push_back(Interval(Pure_interval(Key(1,3), Key(3,9)),'d'));

  Segment_tree_2_type Segment_tree_2(InputList.begin(),InputList.end());

  Interval a=Interval(Pure_interval(Key(3,6), Key(7,12)),'e');
  Segment_tree_2.window_query(a,std::back_inserter(OutputList1));

  std::list<Interval>::iterator j = OutputList1.begin();
  std::cout << "\n window_query (3,6),(7,12)\n";
  while(j!=OutputList1.end()){
    std::cout << (*j).first.first.x() << "-" << (*j).first.second.x() << " "
         << (*j).first.first.y() << "-" << (*j).first.second.y() << std::endl;
    j++;
  }

  Interval b=Interval(Pure_interval(Key(6,10),Key(7,11)), 'f');
  Segment_tree_2.enclosing_query(b,std::back_inserter(OutputList2));
  j = OutputList2.begin();
  std::cout << "\n enclosing_query (6,10),(7,11)\n";
  while(j!=OutputList2.end()){
    std::cout << (*j).first.first.x() << "-" << (*j).first.second.x() << " "
         << (*j).first.first.y() << "-" << (*j).first.second.y() << std::endl;
    j++;
  }
  return 0;
}
```

62.6.2 Example for Segment Tree on Set-like Data

This example illustrates the use of the predefined segment tree on 3-dimensional interval data (with no value associated). After the definition of the traits type and tree type, some intervals are constructed and the tree is build according to the intervals. Then, a window query is performed and the query elements are given out.

```
#include <CGAL/Cartesian.h>
#include <CGAL/Segment_tree_k.h>
#include <CGAL/Range_segment_tree_traits.h>

typedef CGAL::Cartesian<int> K;
```

```
typedef CGAL::Range_segment_tree_set_traits_3<K> Traits;
typedef CGAL::Segment_tree_3<Traits > Segment_tree_3_type;

int main()
{
  typedef Traits::Interval Interval;
  typedef Traits::Key Key;
  std::list<Interval> InputList, OutputList;

  InputList.push_back(Interval(Key(1,5,7), Key(2,7,9)));
  InputList.push_back(Interval(Key(2,7,6), Key(3,8,9)));
  InputList.push_back(Interval(Key(6,9,5), Key(9,13,8)));
  InputList.push_back(Interval(Key(1,3,4), Key(3,9,8)));

  Segment_tree_3_type Segment_tree_3(InputList.begin(),InputList.end());

  Interval a(Key(3,6,5), Key(7,12,8));
  Segment_tree_3.window_query(a,std::back_inserter(OutputList));
  std::list<Interval>::iterator j = OutputList1.begin();
  std::cout << "\n window_query (3,6,5),(7,12,8) \n";
  while(j!=OutputList.end()){
    std::cout << (*j).first.x() << "," << (*j).first.y() << ",";
    std::cout << (*j).first.z() <<", " << (*j).second.x() << ",";
    std::cout << (*j).second.y() << "," << (*j).second.z() << std::endl;
    j++;
  }
}
```

dD Range and Segment Trees Reference Manual

Gabriele Neyer

This chapter presents the CGAL range tree and segment tree data structures.

The range tree is theoretically superior to the Kd-tree, but the latter often seems to perform better. However, the range tree as implemented in CGAL is more flexible than the Kd-tree implementation, in that it enables to layer together range trees and segment trees in the same data structure.

62.7 Classified Reference Pages

Concepts

Traits Classes

Search Structure Classes

62.8 Alphabetical List of Reference Pages

RangeSegmentTreeTraits_k

Definition

A tree traits class gives the range tree and segment tree class the necessary type information of the keys and intervals. Further more, they define function objects that allow to access the keys and intervals, and provide comparison functions that are needed for window queries.

Types

RangeSegmentTreeTraits_k:: Key — The k-dimensional key type.

RangeSegmentTreeTraits_k:: Interval — The k-dimensional interval type.

RangeSegmentTreeTraits_k:: Key_i — The type in dimension i, with $1 \leq i \leq k$.

RangeSegmentTreeTraits_k:: key_i — function object providing an *operator()* that takes an argument of type *Key* and returns a component of type *Key_i*.

RangeSegmentTreeTraits_k:: low_i — function object providing an *operator()* that takes an argument of type *Interval* and returns a component of type *Key_i*.

RangeSegmentTreeTraits_k:: high_i — function object providing an *operator()* that takes an argument of type *Interval* and returns a component of type *Key_i*.

RangeSegmentTreeTraits_k:: compare_i — function object providing an *operator()* that takes two arguments argument a, b of type *Key_i* and returns true if $a < b$, false otherwise.

Example

The following piece of code gives an example of how a traits class might look like, if you have keys that are of the type *int* in the first and that are of the type *double* in the second dimension.

```
class Int_double_tree_traits_2{
 public:
  typedef std::pair<int, double> Key;
  typedef int Key_1;
  typedef double Key_2;
  typedef std::pair<Key,Key> Interval;

  class C_Key_1{
  public:
    Key_1 operator()(const Key& k)
    { return k.first;}
  };
  class C_Key_2{
```

```
  public:
    Key_2 operator()(const Key& k)
    { return k.second;}
  };
  class C_Low_1{
  public:
    Key_1 operator()(const Interval& i)
    { return i.first.first;}
  };
  class C_High_1{
  public:
    Key_1 operator()(const Interval& i)
    { return i.second.first;}
  };
  class C_Low_2{
  public:
    Key_2 operator()(const Interval& i)
    { return i.first.second;}
  };
  class C_High_2{
  public:
    Key_2 operator()(const Interval& i)
    { return i.second.second;}
  };
  class C_Compare_1{
  public:
    bool operator()(Key_1 k1, Key_1 k2)
    { return less<int>()(k1,k2);}
  };
  class C_Compare_2{
  public:
    bool operator()(Key_2 k1, Key_2 k2)
    { return less<double>()(k1,k2);}
  };
  typedef C_Compare_1 compare_1;
  typedef C_Compare_2 compare_2;
  typedef C_Low_1 low_1;
  typedef C_High_1 high_1;
  typedef C_Key_1 key_1;
  typedef C_Low_2 low_2;
  typedef C_High_2 high_2;
  typedef C_Key_2 key_2;
};
```

CGAL::Range_segment_tree_traits_set_2<R>

Definition

The class is a range and segment tree traits class for the 2-dimensional point class from the CGAL kernel. The class is parameterized with a representation class *R*.

#include <CGAL/Range_segment_tree_traits.h>

Types

Point_2<R>	*Key;*
std::pair<Key, Key>	*Interval;*

Class

CGAL::Range_segment_tree_traits_set_3<R>

Definition

The class is a range and segment tree traits class for the 3-dimensional point class from the CGAL kernel. The class is parameterized with a representation class *R*.

#include <CGAL/Range_segment_tree_traits.h>

Types

Point_3<R>	*Key;*
std::pair<Key, Key>	*Interval;*

CGAL::Range_tree_d<Data, Window, Traits>

Class

Types

Range_tree_d<Data, Window, Traits>:: Data	container *Data*.
Range_tree_d<Data, Window, Traits>:: Window	container *Window*.
Range_tree_d<Data, Window, Traits>:: Traits	container *Traits*.

Creation

#include <CGAL/Range_tree_d.h>
Range_tree_d<Data, Window, Traits> r(Tree_base<Data, Window> sublayer_tree);

A range tree is constructed, such that the subtree of each vertex is of the same type prototype *sublayer_tree* is.
We assume that the dimension of the tree is d. This means, that *sublayer_tree* is a prototype of a $d-1$-dimensional tree. All data items of the d-dimensional range tree have container type *Data*. The query window of the tree has container type *Window*. *Traits* provides access to the corresponding data slots of container *Data* and *Window* for the d-th dimension. The traits class *Traits* must at least provide all functions and type definitions as described in, for example, the reference page for *tree_point_traits*. The template class described there is fully generic and should fulfill the most requirements one can have. In order to generate a one-dimensional range tree instantiate *Tree_anchor<Data, Window> sublayer_tree* with the same template parameters (*Data* and *Window*) *Range_tree_d* is defined. In order to construct a two-dimensional range tree, create *Range_tree_d* with a one-dimensional *Range_tree_d* with the corresponding *Traits* class of the first dimension.
Precondition: Traits::Data==Data and Traits::Window==Window.

Operations

template<class ForwardIterator>
bool *r.make_tree(ForwardIterator first, ForwardIterator last)*

The tree is constructed according to the data items in the sequence between the element pointed by iterator *first* and iterator *last*. The data items of the iterator must have type *Data*.

Precondition: This function can only be called once. If it is the first call the tree is build and *true* is returned. Otherwise, nothing is done but a *CGAL warning* is given and *false* returned.

template<class OutputIterator>
OutputIterator *r.window_query(Window win, OutputIterator result)*

All elements that lay inside the d-dimensional interval defined through *win* are placed in the sequence container of *OutputIterator*; the output iterator that points to the last location the function wrote to is returned.

bool *r.is_valid()*

The tree structure is checked. For each vertex the subtree is checked on being valid and it is checked whether the value of the *Key_type* of a vertex corresponds to the highest *Key_type* value of the left subtree.

Protected Operations

bool	*r.is_inside(Window win, Data object)*	returns true, if the data of *object* lies between the start and endpoint of interval *win*. False otherwise.
bool	*r.is_anchor()*	returns false.

Implementation

The construction of a d-dimensional range tree takes $O(n \log n^{d-1})$ time. The points in the query window are reported in time $O(k + \log^d n)$, where k is the number of reported points. The tree uses $O(n \log n^{d-1})$ storage.

Class

CGAL::Range_tree_k<Traits>

Definition

An object of the class is a k-dimensional range tree that can store k-dimensional keys of type *Key*. The class allows to perform window queries on the keys. The class is parameterized with a range tree traits class *Traits* that defines, among other things, the type of the *Key*.

CGAL provides traits class implementations that allow to use the range tree with point classes from the CGAL kernel as keys. These classes are *CGAL::Range_segment_tree_traits_set_2<R>*, *CGAL::Range_segment_tree_traits_set_3<R>*, *CGAL::Range_tree_traits_map_2<R>* and *CGAL::Range_tree_traits_map_3<R>*. The concept RangeSegmentTreeTraits_d defines the requirements that range tree traits classes must fulfill. This allows the advanced user to develop further range tree traits classes.

#include <CGAL/Range_tree_k.h>

Types

Range_tree_k<Traits>:: Traits — the type of the range tree traits class.

typedef Traits::Key Key;

typedef Traits::Interval Interval;

Creation

Range_tree_k<Traits> R; — Introduces an empty range tree *R*.

template < class ForwardIterator >
Range_tree_k<Traits> R(ForwardIterator first, ForwardIterator last);

Introduces a range tree *R* and initializes it with the data in the range *[first, last)*.
Precondition: *value_type(first) == Traits::Key*.

Operations

template < class ForwardIterator >
void R.make_tree(ForwardIterator first, ForwardIterator last)

Introduces a range tree *R* and initializes it with the data in the range *[first, last)*. This function can only be applied once on an empty range tree.
Precondition: *value_type(first) == Traits::Key*.

template < class OutputIterator >
OutputIterator *R.window_query(Interval window, OutputIterator out)*

writes all data that are in the interval *window* to the container where *out* points to, and returns an output iterator that points to the last location the function wrote to.
Precondition: *value_type(out) == Traits::Key*.

Example

The following example program uses the predefined *Range_tree_2* data structure together with the predefined traits class *Range_tree_map_traits_2* which has two template arguments specifying the type of the point data in each dimension (*CGAL::Cartesian<double>*) and the value type of the 2-dimensional point data (*char*). Therefore the *Range_tree_2* is defined on 2-dimensional point data (*CGAL::Point_2<Cartesian<double> >*) each of which is associated with a character. Then, a few data items are created and put into a list. After that the tree is constructed according to that list, a window query is performed, and the query elements are given out.

```
#include <CGAL/Cartesian.h>
#include <CGAL/Range_segment_tree_traits.h>
#include <CGAL/Range_tree_k.h>

typedef CGAL::Cartesian<double> K;
typedef CGAL::Range_tree_map_traits_2<K, char> Traits;
typedef CGAL::Range_tree_2<Traits> Range_tree_2_type;

int main()
{
  typedef Traits::Key Key;
  typedef Traits::Interval Interval;

  std::vector<Key> InputList, OutputList;
  InputList.push_back(Key(K::Point_2(8,5.1), 'a'));
  InputList.push_back(Key(K::Point_2(1,1.1), 'b'));
  InputList.push_back(Key(K::Point_2(3,2.1), 'c'));

  Range_tree_2_type Range_tree_2(InputList.begin(),InputList.end());
  Interval win(Interval(K::Point_2(4,8.1), K::Point_2(5,8.2)));
  std::cout << "\n Window Query:\n ";
  Range_tree_2.window_query(win, std::back_inserter(OutputList));
  std::vector<Key>::iterator current=OutputList.begin();
  while(current!=OutputList.end()){
    std::cout << (*current).first.x() << "," << (*current).first.y()
         << ":" << (*current++).second << std::endl;
  }
}
```

Class

CGAL::Range_tree_traits_map_2<R,T>

Definition

The class is a range tree traits class for the 2-dimensional point class from the CGAL kernel, where data of type T is associated to each key. The class is parameterized with a representation class R and the type of the associated data T.

#include <CGAL/Range_segment_tree_traits.h>

Types

std::pair<R::Point_2,T>

Key;

std::pair<R::Point_2, R::Point_2 >

Interval;

CGAL::Range_tree_traits_map_3<R,T>

Definition

The class is a range and segment tree traits class for the 3-dimensional point class from the CGAL kernel, where data of type T is associated to each key. The class is parameterized with a representation class R and the type of the associated data T.

#include <CGAL/Range_segment_tree_traits.h>

Types

std::pair<R::Point_3,T>

Key;

std::pair<R::Point_3, R::Point_3>

Interval;

Class

CGAL::Segment_tree_d<Data, Window, Traits>

Types

Segment_tree_d<Data, Window, Traits>:: Data

container *Data*.

Segment_tree_d<Data, Window, Traits>:: Window

container *Window*.

Segment_tree_d<Data, Window, Traits>:: Traits

class *Traits*.

Creation

#include <CGAL/Segment_tree_d.h>
Segment_tree_d<Data, Window, Traits> s(Tree_base<Data, Window> sublayer_tree);

A segment tree is defined, such that the subtree of each vertex is of the same type prototype *sublayer_tree* is.
We assume that the dimension of the tree is d. This means, that *sublayer_tree* is a prototype of a $d-1$-dimensional tree. All data items of the d-dimensional segment tree have container type *Data*. The query window of the tree has container type *Window*. *Traits* provides access to the corresponding data slots of container *Data* and *Window* for the d-th dimension. The traits class *Traits* must at least provide all functions and type definitions described, for example, in the reference page for *tree_point_traits*. The template class described there is fully generic and should fulfill the most requirements one can have. In order to generate a one-dimensional segment tree instantiate *Tree_anchor<Data, Window> sublayer_tree* with the same template parameters *Data* and *Window Segment_tree_d* is defined. In order to construct a two-dimensional segment tree, create *Segment_tree_d* with a one-dimensional *Segment_tree_d* with the corresponding *Traits* of the first dimension.
Precondition: Traits::Data==Data and Traits::Window==Window.

Operations

bool *s.make_tree(In_it first, In_it last)*

The tree is constructed according to the data items in the sequence between the element pointed by iterator *first* and iterator *last*.

Precondition: This function can only be called once. If it is the first call the tree is build and *true* is returned. Otherwise, nothing is done but a *CGAL warning* is given and *false* returned.

OutputIterator *s.window_query(Window win, OutputIterator result)*

win$= [a_1,b_1),\ldots,[a_d,b_d)$, $a_i,b_i \in T_i$, $1 \leq i \leq d$. All elements that intersect the associated d-dimensional interval of *win* are placed in the associated sequence container of *OutputIterator* and returns an output iterator that points to the last location the function wrote to. In order to perform an inverse range query, a range query of ε width has to be performed.

OutputIterator *s.enclosing_query(Window win, OutputIterator result)*

All elements that enclose the associated d-dimensional interval of *win* are placed in the associated sequence container of *OutputIterator* and returns an output iterator that points to the last location the function wrote to.

bool *s.is_valid()* The tree structure is checked. For each vertex either the sublayer tree is a tree anchor, or it stores a (possibly empty) list of data items. In the first case, the sublayer tree of the vertex is checked on being valid. In the second case, each data item is checked weather it contains the associated interval of the vertex and does not contain the associated interval of the parent vertex or not. True is returned if the tree structure is valid, false otherwise.

Protected Operations

bool *s.is_inside(Window win, Data object)*

returns true, if the interval of *object* is contained in the interval of *win*. False otherwise.

bool *s.is_anchor()* returns false.

Implementation

A d-dimensional segment tree is constructed in $O(n \log n^d)$ time. An inverse range query is performed in time $O(k + \log^d n)$, where k is the number of reported intervals. The tree uses $O(n \log n^d)$ storage.

CGAL::Segment_tree_k<Traits>

Definition

An object of the class is a *k*-dimensional segment tree that can store k-dimensional intervals of type *Interval*. The class allows to perform window queries, enclosing queries, and inverse range queries on the keys. The class is parameterized with a segment tree traits class *Traits* that defines, among other things, the type of the *Interval*. In order to perform an inverse range query, a range query of ε width has to be performed. We prefered not to offer an extra function for this sort of query, since the inverse range query is a special case of the range query. Furthermore, offering an inverse range query in the segment tree class implies offering this function also in the range tree class and having an extra item in the traits class that accesses the inverse range query point.

CGAL provides traits class implementations that allow to use the segment tree with point classes from the CGAL kernel as keys. These classes are *CGAL::Range_segment_tree_traits_set_2<R>*, *CGAL::Range_segment_tree_traits_set_3<R>*, *CGAL::Segment_tree_traits_map_2<R>* and *CGAL::Segment_tree_traits_map_3<R>*. The concept RangeSegmentTreeTraits_d defines the requirements that segment tree traits classes must fulfill. This allows the advanced user to develop further segment tree traits classes.

#include <CGAL/Segment_tree_k.h>

Types

Segment_tree_k<Traits>:: Traits — the type of the segment tree traits class.

typedef Traits::Key *Key;*

typedef Traits::Interval

Interval;

Creation

Segment_tree_k<Traits> S; — Introduces an empty segment tree *S*.

template < class ForwardIterator >
Segment_tree_k<Traits> S(ForwardIterator first, ForwardIterator last);

Introduces a segment tree *S* and initializes it with the data in the range *[first, last)*.
Precondition: *value_type(first) == Traits::Interval*.

Operations

template < class ForwardIterator >
void *S.make_tree(ForwardIterator first, ForwardIterator last)*

Introduces a segment tree *S* and initializes it with the data in the range *[first, last)*. This function can only be applied once on an empty segment tree.
Precondition: *value_type(first) == Traits::Interval*.

template < class OutputIterator >
OutputIterator *S.window_query(Interval window, OutputIterator out)*

writes all intervals that have non empty intersection with interval *window* to the container where *out* points to, and returns an output iterator that points to the last location the function wrote to.
Precondition: *value_type(out) == Traits::Interval*.

template < class OutputIterator >
OutputIterator *S.enclosing_query(Interval window, OutputIterator out)*

writes all intervals that enclose in the interval *window* to the container where *out* points to, and returns an output iterator that points to the last location the function wrote to.
Precondition: *value_type(out) == Traits::Interval*.

Example

This example illustrates the use of the predefined segment tree on 3-dimensional interval data (with no value associated). After the definition of the traits type and tree type, some intervals are constructed and the tree is build according to the intervals. Then, a window query is performed and the query elements are given out.

```
#include <CGAL/Cartesian.h>
#include <CGAL/Segment_tree_k.h>
#include <CGAL/Range_segment_tree_traits.h>

typedef CGAL::Cartesian<int> K;
typedef CGAL::Range_segment_tree_set_traits_3<K> Traits;
typedef CGAL::Segment_tree_3<Traits> Segment_tree_3_type;

int main()
{
  typedef Traits::Interval Interval;
  typedef Traits::Key Key;
  std::list<Interval> InputList, OutputList;

  InputList.push_back(Interval(Key(1,5,7), Key(2,7,9)));
  InputList.push_back(Interval(Key(2,7,6), Key(3,8,9)));
  InputList.push_back(Interval(Key(6,9,5), Key(9,13,8)));
  InputList.push_back(Interval(Key(1,3,4), Key(3,9,8)));

  Segment_tree_3_type Segment_tree_3(InputList.begin(),InputList.end());

  Interval a(Key(3,6,5), Key(7,12,8));
  Segment_tree_3.window_query(a,std::back_inserter(OutputList));
  std::list<Interval>::iterator j = OutputList1.begin();
  std::cout << "\n window_query (3,6,5),(7,12,8) \n";
  while(j!=OutputList.end()){
    std::cout << (*j).first.x() << "," << (*j).first.y() << ",";
    std::cout << (*j).first.z() <<", " << (*j).second.x() << ",";
    std::cout << (*j).second.y() << "," << (*j).second.z() << std::endl;
```

```
      j++;
    }
  }
```

CGAL::Segment_tree_traits_map_2<R,T>

Definition

The class is a segment tree traits class for the 2-dimensional point class from the CGAL kernel, where data of type T is associated to each interval. The class is parameterized with a representation class R and the type of the associated data T.

#include <CGAL/Range_segment_tree_traits.h>

Types

R::Point_2	*Key;*
std::pair<std::pair<Key,Key>,T>	*Interval;*

Class

Class

CGAL::Segment_tree_traits_map_3<R,T>

Definition

The class is a segment tree traits class for the 3-dimensional point class from the CGAL kernel, where data of type T is associated to each interval. The class is parameterized with a representation class R and the type of the associated data T.

#include <CGAL/Range_segment_tree_traits.h>

Types

std::pair<R::Point_3 >

Key;

std::pair<std::pair<Key, Key>,T>

Interval;

Sublayer

Definition

Defines the requirements that a *Sublayer_type* of class *Range_tree_d* or *Segment_tree_d* has to fulfill.

First of all, the class has to be derived from the abstract base class *Tree_base* and therefore has to provide methods *make_tree*, *window_query*, *enclosing_query* and *is_inside* with the same parameter types as the instantiated class *Range_tree_d* or *Segment_tree_d*, respectively. Furthermore a method *bool is_anchor()* has to be provided. If the *Sublayer_type* class builds a recursion anchor for class *Segment_tree_d*, this function is expected to return *true*, *false* otherwise.

Such a recursion anchor class is provided by the class class. *Tree_anchor<Data, Window>*.

CGAL::tree_interval_traits

Definition

tree_interval_traits is a template class that provides an interface to data items. It is similar to *tree_point_traits*, except that it provides access to two data slots of the same type of each container class (*Data, Window*) instead of providing access to one data slot of container class *Data* and two data slots of class *Window*.

#include <CGAL/Tree_traits.h>

Types

typedef
tree_interval_traits<Data, Window, Key, Data_left_func, Data_right_func, Window_left_func, Window_right_func, Compare> Interval_traits;

tree_interval_traits:: Data	the container *Data* — the data type. It may consist of several data slots. Two of these data slots have to be of type *Key*.
tree_interval_traits:: Window	the container *Window* — the query window type. It may consist of several data slots. Two of these data slots have to be of type *Key*.
tree_interval_traits:: Key	the type *Key* of the data slot this traits class provides access to.
tree_interval_traits:: Data_left_func	*Data_left_func* is a function object providing an *operator()* that takes an argument of type *Data* and returns a (the left) component of type *Key*.
tree_interval_traits:: Data_right_func	*Data_right_func* is a function object providing an *operator()* that takes an argument of type *Data* and returns a (the right) component of type *Key*.
tree_interval_traits:: Window_left_func	*Window_left_func* is a function objects that allow to access the left data slot of container *Window* which has type *Key*
tree_interval_traits:: Window_right_func	*Window_right_func* is a function objects that allow to access the right data slot of container *Window* which has type *Key*
tree_interval_traits:: Compare	defines a comparison relation which must define a strict ordering of the objects of type *Key*. If defined, *less<Key>* is sufficient.

Creation

tree_interval_traits<Data, Window, Key, Data_left_func, Data_right_func, Window_left_func, Window_right_func, Compare> d();

Generation of a *tree_point_traits* instance. It is a template class that provides an interface to data items.

Operations

Key	*d.get_left(Data d)*	The data slot of the data item of *d* of type *Key* is accessed by function object *Data_left_func*.
Key	*d.get_right(Data d)*	The data slot of the data item of *d* of type *Key* is accessed by function object *Data_right_func*.
Key	*d.get_left_win(Window w)*	
		The data slot of the data item of *w* of type *Key* is accessed by function object *Window_left_func*.
Key	*d.get_right_win(Window w)*	
		The data slot of the data item of *w* of type *Key* is accessed by function object *Window_right_func*.
static bool	*d.comp(Key& key1, Key& key2)*	
		returns Compare(key1, key2).

Class

CGAL::tree_point_traits

Definition

tree_point_traits is a template class that provides an interface to data items.

#include <CGAL/Tree_traits.h>

Types

typedef
tree_point_traits<Data, Window, Key, Data_func, Window_left_func, Window_right_func, Compare> Point_traits;

tree_point_traits:: Data	the container *Data* — defines the Data type. It may consist of several data slots. One of these data slots has to be of type *Key*.
tree_point_traits:: Window	the container *Window* — defines the type of the query rectangle. It may consist of several data slots. Two of these data slots has to be of type *Key*
tree_point_traits:: Key	the type *Key* of the data slot this traits class provides access to.
tree_point_traits:: Data_func	*Data_func* is a function object providing an *operator()* that takes an argument of type *Data* and returns a component of type *Key*.
tree_point_traits:: Window_left_func	*Window_left_func* is a function objects that allow to access the left data slot of container *Window* which has type *Key*
tree_point_traits:: Window_right_func	*Window_right_func* is a function objects that allow to access the right data slot of container *Window* which has type *Key*
tree_point_traits:: Compare	defines a comparison relation which must define a strict ordering of the objects of type *Key*. If defined, *less<Key>* is sufficient.

Creation

tree_point_traits<Data, Window, Key, Data_func, Window_left_func, Window_right_func, Compare> d();

Generation of a *tree_point_traits* instance. It is a template class that provides an interface to data items.

Operations

Key	*d.get_key(Data d)*	The data slot of the data item of *d* of type *Key* is accessed by function object *Data_func*.
Key	*d.get_left(Window w)*	The data slot of the data item of *w* of type *Key* is accessed by function object *Window_left_func*.
Key	*d.get_right(Window w)*	The data slot of the data item of *w* of type *Key* is accessed by function object *Window_right_func*.
static bool	*d.comp(Key& key1, Key& key2)*	returns *Compare(key1, key2)*.
static bool	*d.key_comp(Data& data1, Data& data2)*	returns *Compare(get_key(data1), get_key(data2))*.

Class

CGAL::Tree_anchor<Data, Window>

Definition

Tree_anchor is also derived from *Tree_base*. Therefore, it provides the same methods as *Range_tree_d* and *Segment_tree_d*, but does nothing; it can be used as a recursion anchor for those classes. Therefore, instantiate *Sublayer_type* of *Range_tree_d* (*Segment_tree_d* respectively) with *Tree_anchor* and the container classes for the data items (*Data* and *Window*).

Definition

Tree_anchor<Data, Window>:: Data — container *Data*.

Tree_anchor<Data, Window>:: Window — container *Window*.

Creation

#include <CGAL/Tree_base.h>

Tree_anchor<Data, Window> a;

Operations

template<class OutputIterator>
OutputIterator *a.window_query(Window win, OutputIterator result)*

template<class OutputIterator>
OutputIterator *a.enclosing_query(Window win, OutputIterator result)*

bool *a.is_valid()* returns true;

Protected Operations

bool *a.is_inside(Window win, Data object)*

returns true.

bool *a.is_anchor()* returns true.

Example

The following figures show a number of rectangles and a 2-dimensional segment tree built on them.

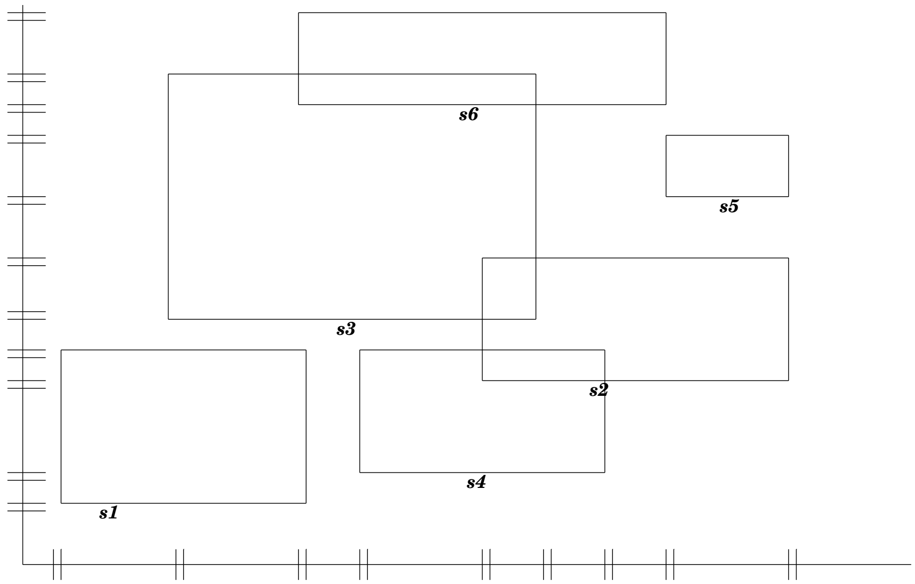


Figure 62.6: Two dimensional interval data.

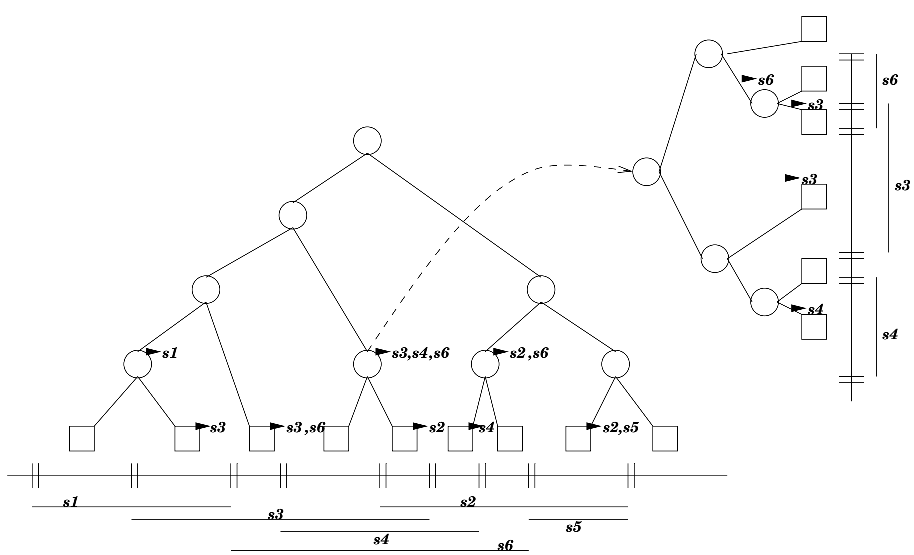
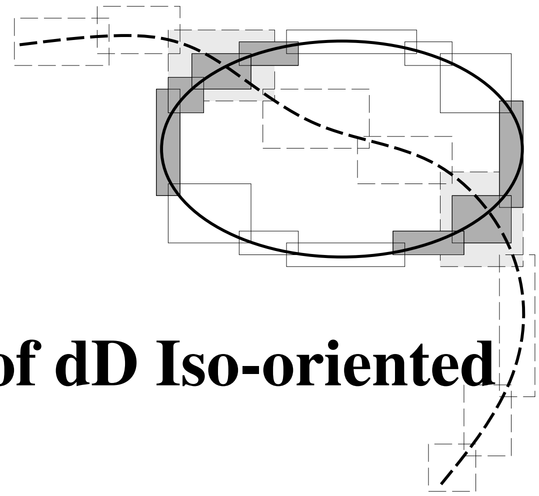


Figure 62.7: Two dimensional segment tree according to the interval data of Figure 62.6.



Chapter 63

Intersecting Sequences of dD Iso-oriented Boxes

Lutz Kettner, Andreas Meyer, and Afra Zomorodian

Contents

63.1 Introduction

Simple questions on geometric primitives, such as intersection and distance computations, can themselves become quite expensive if the primitives are not so simple anymore, for example, three-dimensional triangles and facets of polyhedral surfaces. Thus algorithms operating on these primitives tend to be slow in practice. A common (heuristic) optimization approximates the geometric primitives with their axis-aligned bounding boxes, runs a suitable modification of the algorithm on the boxes, and whenever a pair of boxes has an interesting interaction[1], only then the exact answer is computed on the complicated geometric primitives contained in the boxes.

[1] Boxes represent volumes or point-sets. So, *intersection* means intersection of the point-set enclosed by the box and not only intersection of the boundary, of course.

We provide an efficient algorithm [ZE02] for finding all intersecting pairs for large numbers of iso-oriented boxes, i.e., typically these will be such bounding boxes of more complicated geometries. One immediate application of this algorithm is the detection of all intersections (and self-intersections) for polyhedral surfaces, i.e., applying the algorithm on a large set of triangles in space, we give an example program later in this chapter. Not so obvious applications are proximity queries and distance computations among such surfaces, see Section 63.10 for an example and [ZE02] for more details.

63.2 Definition

A d-dimensional iso-oriented box is defined as the Cartesian product of d intervals. We call the box *half-open* if the d intervals $\{[lo_i, hi_i) \,|\, 0 \le i < d\}$ are half-open intervals, and we call the box *closed* if the d intervals $\{[lo_i, hi_i] \,|\, 0 \le i < d\}$ are closed intervals. Note that closed boxes support zero-width boxes and they can intersect at their boundaries, while non-empty half-open boxes always have a positive volume and they only intersect iff their interiors overlap. The distinction between closed and half-open boxes does not require a different representation of boxes, just a different interpretation when comparing boxes, which is selected with the two possible values for the *topology* parameter:

- *CGAL::Box_intersection_d::HALF_OPEN* and
- *CGAL::Box_intersection_d::CLOSED*.

The number type of the interval boundaries must be one of the built-in types `int`, `unsigned int`, `double` or `float`.

In addition, a box has an unique *id*-number. It is used to order boxes consistently in each dimension even if boxes have identical coordinates. In consequence, the algorithm guarantees that a pair of intersecting boxes is reported only once. Note that boxes with equal *id*-number are not reported since they obviously intersect trivially.

The box intersection algorithm comes in two flavors: One algorithm works on a single sequence of boxes and computes all pairwise intersections, which is called the *complete* case, and used, for example, in the self-intersection test. The other algorithm works on two sequences of boxes and computes the pairwise intersections between boxes from the first sequence with boxes from the second sequence, which is called the *bipartite* case. For each pairwise intersection found a callback function is called with two arguments; the first argument is a box from the first sequence and the second argument a box from the second sequence. In the complete case, the second argument is a box from an internal copy of the first sequence.

63.3 Software Design

The box intersection algorithm is implemented as a family of generic functions; the functions for the complete case accept one iterator range, and the functions for the bipartite case accept two iterator ranges. The callback function for reporting the intersecting pairs is provided as a template parameter of the *BinaryFunction* concept. The two principle function calls utilizing all default arguments look as follows:

#include <CGAL/box_intersection_d.h>

template< class RandomAccessIterator, class Callback >
void box_intersection_d(RandomAccessIterator begin, RandomAccessIterator end, Callback callback)

template< class RandomAccessIterator1, class RandomAccessIterator2, class Callback >
void box_intersection_d(RandomAccessIterator1 begin1,
RandomAccessIterator1 end1,
RandomAccessIterator2 begin2,
RandomAccessIterator2 end2,
Callback callback)

Additional parameters to the functions calls are a *cutoff* value to adjust performance trade-offs, and a *topology* parameter selecting between topologically closed boxes (the default) and topologically half-open boxes.

The algorithm reorders the boxes in the course of the algorithm. Now, depending on the size of a box it can be faster to copy the boxes, or to work with pointers to boxes and copy only pointers. We offer automatic support for both options. To simplify the description, let us call the *value_type* of the iterator ranges *box handle*. The *box handle* can either be our box type itself or a pointer (or const pointer) to the box type; these choices represent both options from above.

In general, the algorithms treat the box type as opaque type and just assume that they are models of the *Assignable* concept, so that the algorithms can modify the input sequences and reorder the boxes. The access to the box dimension and box coordinates is mediated with a traits class of the *BoxIntersectionTraits_d* concept. A default traits class is provided that assumes that the box type is a model of the *BoxIntersectionBox_d* concept and that the box handle, i.e., the iterators value type, is identical to the box type or a pointer to the box type (see the previous paragraph for the value versus pointer nature of the box handle).

Two implementations of iso-oriented boxes are provided; *CGAL::Box_intersection_d::Box_d* as a plain box, and *CGAL::Box_intersection_d::Box_with_handle_d* as a box plus a handle that can be used to point to the full geometry that is approximated by the box. Both implementations have template parameters for the number type used for the interval bounds, for the fixed dimension of the box, and for a policy class [Ale01] selecting among several solutions for providing the *id*-number.

The function signatures for the bipartite case look as follows. The signatures for the complete case with the *box_self_intersection_d* function look the same except for the single iterator range.

#include <CGAL/box_intersection_d.h>

template< class RandomAccessIterator1, class RandomAccessIterator2, class Callback >
void box_intersection_d(RandomAccessIterator1 begin1,
RandomAccessIterator1 end1,
RandomAccessIterator2 begin2,
RandomAccessIterator2 end2,
Callback callback,
std::ptrdiff_t cutoff = 10,
Box_intersection_d::Topology topology = Box_intersection_d::CLOSED,
Box_intersection_d::Setting setting = Box_intersection_d::BIPARTITE)

template< class RandomAccessIterator1, class RandomAccessIterator2, class Callback, class BoxTraits >
void box_intersection_d(RandomAccessIterator1 begin1,
RandomAccessIterator1 end1,
RandomAccessIterator2 begin2,
RandomAccessIterator2 end2,
Callback callback,
BoxTraits box_traits,
std::ptrdiff cutoff = 10,
Box_intersection_d::Topology topology = Box_intersection_d::CLOSED,
Box_intersection_d::Setting setting = Box_intersection_d::BIPARTITE)

63.4 Minimal Example for Intersecting Boxes

The box implementation provided with *CGAL::Box_intersection_d::Box_d<double,2>* has a dedicated constructor for the CGAL bounding box type *CGAL::Bbox_2* (similar for dimension 3). We use this in our minimal example to create easily nine two-dimensional *boxes* in a grid layout of 3×3 boxes. Additionally we pick the center box and the box in the upper-right corner as our second box sequence *query*.

The default policy of the box type implements the *id*-number with an explicit counter in the boxes, which is the default choice since it always works, but it costs space that could potentially be avoided, see the example in the next section. We use the *id*-number in our callback function to report the result of the intersection algorithm. The result will be that the first *query* box intersects all nine *boxes* and the second *query* box intersects the four boxes in the upper-right quadrant. See Section 63.7 for the change of the *topology* parameter and its effect.

```
#include <CGAL/box_intersection_d.h>
#include <CGAL/Bbox_2.h>
#include <iostream>

typedef CGAL::Box_intersection_d::Box_d<double,2> Box;
typedef CGAL::Bbox_2                              Bbox;
                                                     // 9 boxes of a grid
Box boxes[9] = { Bbox( 0,0,1,1), Bbox( 1,0,2,1), Bbox( 2,0,3,1), // low
                 Bbox( 0,1,1,2), Bbox( 1,1,2,2), Bbox( 2,1,3,2), //
   middle
                 Bbox( 0,2,1,3), Bbox( 1,2,2,3), Bbox( 2,2,3,3)};// upper
// 2 selected boxes as query; center and upper right
Box query[2] = { Bbox( 1,1,2,2), Bbox( 2,2,3,3)};

void callback( const Box& a, const Box& b ) {
    std::cout << "box " << a.id() << " intersects box " << b.id() <<
   std::endl;
}
int main() {
    CGAL::box_intersection_d( boxes, boxes+9, query, query+2, callback);
    return 0;
}
```

File: examples/Box_intersection_d/minimal.cpp

63.5 Example for Finding Intersecting 3D Triangles

The conventional application of the axis-aligned box intersection algorithm will start from complex geometry, here 3D triangles, approximate them with their bounding box, compute the intersecting pairs of boxes, and check only for those if the original triangles intersect as well.

We start in the *main* function and create ten triangles with endpoints chosen randomly in a cube $[-1,+1)^3$. We store the triangles in a vector called *triangles*.

Next we create a vector for the bounding boxes of the triangles called *boxes*. For the boxes we choose the type *Box_with_handle_d<double,3,Iterator>* that works nicely together with the CGAL bounding boxes of type *CGAL::Bbox_3*. In addition, each box stores the iterator to the corresponding triangle.

The default policy of this box type uses for the *id*-number the address of the value of the iterator, i.e., the address of the triangle. This is a good choice that works correctly iff the boxes have unique iterators, i.e., there is a one-to-one mapping between boxes and approximated geometry, which is the case here. It saves us the extra space that was needed for the explicit *id*-number in the previous example.

We run the self intersection algorithm with the *report_inters* function as callback. This callback reports the intersecting boxes. It uses the *handle* and the global *triangles* vector to calculate the triangle numbers. Then it checks the triangles themselves for intersection and reports if not only the boxes but also the triangles intersect. We take some precautions before the intersection test in order to avoid problems, although unlikely, with degenerate triangles that we might have created with the random process.

This example can be easily extended to test polyhedral surfaces of the *Polyhedron_3* class for (self-) intersections. The main difference are the numerous cases of incidences between triangles in the polyhedral surface that should not be reported as intersections, see the `examples/Polyhedron/polyhedron_self_intersection.cpp` example program in the CGAL distribution.

```
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/intersections.h>
#include <CGAL/point_generators_3.h>
#include <CGAL/Bbox_3.h>
#include <CGAL/box_intersection_d.h>
#include <CGAL/function_objects.h>
#include <CGAL/Join_input_iterator.h>
#include <CGAL/algorithm.h>
#include <vector>

typedef CGAL::Exact_predicates_inexact_constructions_kernel   Kernel;
typedef Kernel::Point_3                                        Point_3;
typedef Kernel::Triangle_3                                     Triangle_3;
typedef std::vector<Triangle_3>                                Triangles;
typedef Triangles::iterator                                    Iterator;
typedef CGAL::Box_intersection_d::Box_with_handle_d<double,3,Iterator>
    Box;

Triangles triangles; // global vector of all triangles

// callback function that reports all truly intersecting triangles
void report_inters( const Box& a, const Box& b) {
    std::cout << "Box " << (a.handle() - triangles.begin()) << " and "
              << (b.handle() - triangles.begin()) << " intersect";
    if ( ! a.handle()->is_degenerate() && ! b.handle()->is_degenerate()
         && CGAL::do_intersect( *(a.handle()), *(b.handle()))) {
        std::cout << ", and the triangles intersect also";
    }
    std::cout << '.' << std::endl;
}

int main() {
    // Create 10 random triangles
    typedef CGAL::Random_points_in_cube_3<Point_3>          Pts;
    typedef CGAL::Creator_uniform_3< Point_3, Triangle_3>    Creator;
    typedef CGAL::Join_input_iterator_3<Pts,Pts,Pts,Creator>
   Triangle_gen;
    Pts     points( 1); // in centered cube [-1,1)^3
```

```
    Triangle_gen triangle_gen( points, points, points);
    CGAL::cpp0x::copy_n( triangle_gen, 10,
   std::back_inserter(triangles));

    // Create the corresponding vector of bounding boxes
    std::vector<Box> boxes;
    for ( Iterator i = triangles.begin(); i != triangles.end(); ++i)
        boxes.push_back( Box( i->bbox(), i));

    // Run the self intersection algorithm with all defaults
    CGAL::box_self_intersection_d( boxes.begin(), boxes.end(),
   report_inters);
    return 0;
}
```

File: examples/Box_intersection_d/triangle_self_intersect.cpp

63.6 Example for Using Pointers to Boxes

We modify the previous example, finding intersecting 3D triangles, and add an additional vector *ptr* that stores pointers to the bounding boxes, so that the intersection algorithm will work on a sequence of pointers and not on a sequence of boxes. The change just affects the preparation of the additional vector and the call of the box intersection function. The box intersection function (actually its default traits class) detects automatically that the value type of the iterators is a pointer type and not a class type.

```
    // Create the corresponding vector of pointers to bounding boxes
    std::vector<Box *> ptr;
    for ( std::vector<Box>::iterator i = boxes.begin(); i != boxes.end();
    ++i)
        ptr.push_back( &*i);

    // Run the self intersection algorithm with all defaults on the
    // indirect pointers to bounding boxes. Avoids copying the boxes.
    CGAL::box_self_intersection_d( ptr.begin(), ptr.end(),
   report_inters);
```

In addition, the callback function *report_inters* needs to be changed to work with pointers to boxes. The full example program looks as follows:

```
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/intersections.h>
#include <CGAL/point_generators_3.h>
#include <CGAL/Bbox_3.h>
#include <CGAL/box_intersection_d.h>
#include <CGAL/function_objects.h>
#include <CGAL/Join_input_iterator.h>
#include <CGAL/algorithm.h>
#include <vector>

typedef CGAL::Exact_predicates_inexact_constructions_kernel   Kernel;
typedef Kernel::Point_3                                       Point_3;
```

```
typedef Kernel::Triangle_3                                          Triangle_3;
typedef std::vector<Triangle_3>                                     Triangles;
typedef Triangles::iterator                                         Iterator;
typedef CGAL::Box_intersection_d::Box_with_handle_d<double,3,Iterator>
   Box;

Triangles triangles; // global vector of all triangles

// callback function that reports all truly intersecting triangles
void report_inters( const Box* a, const Box* b) {
    std::cout << "Box " << (a->handle() - triangles.begin()) << " and "
              << (b->handle() - triangles.begin()) << " intersect";
    if ( ! a->handle()->is_degenerate() && ! b->handle()->is_degenerate()
         && CGAL::do_intersect( *(a->handle()), *(b->handle()))) {
        std::cout << ", and the triangles intersect also";
    }
    std::cout << '.' << std::endl;
}

int main() {
    // Create 10 random triangles
    typedef CGAL::Random_points_in_cube_3<Point_3>          Pts;
    typedef CGAL::Creator_uniform_3< Point_3, Triangle_3>    Creator;
    typedef CGAL::Join_input_iterator_3<Pts,Pts,Pts,Creator>
   Triangle_gen;
    Pts    points( 1); // in centered cube [-1,1)^3
    Triangle_gen triangle_gen( points, points, points);
    CGAL::cpp0x::copy_n( triangle_gen, 10,
   std::back_inserter(triangles));

    // Create the corresponding vector of bounding boxes
    std::vector<Box> boxes;
    for ( Iterator i = triangles.begin(); i != triangles.end(); ++i)
        boxes.push_back( Box( i->bbox(), i));

    // Create the corresponding vector of pointers to bounding boxes
    std::vector<Box *> ptr;
    for ( std::vector<Box>::iterator i = boxes.begin(); i != boxes.end();
    ++i)
        ptr.push_back( &*i);

    // Run the self intersection algorithm with all defaults on the
    // indirect pointers to bounding boxes. Avoids copying the boxes.
    CGAL::box_self_intersection_d( ptr.begin(), ptr.end(),
   report_inters);
    return 0;
}
```

File: examples/Box_intersection_d/triangle_self_intersect_pointers.cpp

A note on performance: The algorithm sorts and partitions the input sequences. It is clearly costly to copy a large box compared to a simple pointer. However, the algorithm benefits from memory locality in the later stages when it copies the boxes, while the pointers would refer to boxes that become wildly scattered in memory. These two effects, copying costs and memory locality, counteract each other. For small box sizes, i.e., small dimension, memory locality wins and one should work with boxes, while for larger box sizes one should work with pointers. The exact threshold depends on the memory hierarchy (caching) of the hardware platform and the size of the boxes, most notably the type used to represent the box coordinates. A concrete example; on a laptop with an Intel Mobile Pentium4 running at 1.80GHz with 512KB cache and 254MB main memory under Linux this version with pointers was 20% faster than the version above that copies the boxes for 10000 boxes, but the picture reversed for 100000 boxes, where the version above that copies the boxes becomes 300% faster.

Note that switching to the built-in type *float* is supported by the box intersection algorithm, but the interfacing with the CGAL bounding box *CGAL::Bbox_3* would not be that easy. In particular, just converting from the *double* to the *float* representation incurs rounding that needs to be controlled properly, otherwise the box might shrink and one might miss intersections.

63.7 Example Using the *topology* and the *cutoff* Parameters

Boxes can be interpreted by the box intersection algorithm as closed or as half-open boxes, see also Section 63.2. Closed boxes support zero-width boxes and they can intersect at their boundaries, while half-open boxes always have a positive volume and they only intersect iff their interiors overlap. The choice between closed or half-open boxes is selected with the *topology* parameter and its two values:

- *CGAL::Box_intersection_d::HALF_OPEN* and
- *CGAL::Box_intersection_d::CLOSED*.

The example program uses a two-dimensional box with *int* coordinates and *id*-numbers that are by default explicitly stored. We create the same boxes as in the minimal example in Section 63.4. We create a 3×3 grid of *boxes*, and two boxes for the *query* sequence, namely the box at the center and the box from the upper-right corner of the grid.

We write a more involved callback function object *Report* that stores an output iterator and writes the *id*-number of the box in the first argument to the output iterator. We also provide a small helper function *report* that simplifies the use of the function object.

We call the box intersection algorithm twice; once for the default *topology*, which is the closed box topology, and once for the half-open box topology. We sort the resulting output for better readability and verify its correctness with the *check1* and *check2* data. For the closed box topology, the center box in *query* intersects all *boxes*, and the upper-right box in *query* intersects the four boxes of the upper-right quadrant in *boxes*. Almost all intersections are with the box boundaries, thus, for the half-open topology only one intersection remains per *query* box, namely its corresponding box in *boxes*. So, the output of the algorithm will be:

```
0 1 2 3 4 4 5 5 6 7 7 8 8
4 8
```

For the second box intersection function call we have to specify the *cutoff* parameter explicitly. See the Section 63.8 below for a detailed discussion.

```
#include <CGAL/box_intersection_d.h>
#include <vector>
#include <algorithm>
```

```
#include <iterator>
#include <cassert>

typedef CGAL::Box_intersection_d::Box_d<int,2> Box;

// coordinates for 9 boxes of a grid
int p[9*4]   = { 0,0,1,1,  1,0,2,1,  2,0,3,1, // lower
                 0,1,1,2,  1,1,2,2,  2,1,3,2, // middle
                 0,2,1,3,  1,2,2,3,  2,2,3,3};// upper
// 9 boxes
Box boxes[9] = { Box( p,    p+ 2),  Box( p+ 4, p+ 6),  Box( p+ 8, p+10),
                 Box( p+12, p+14),  Box( p+16, p+18),  Box( p+20, p+22),
                 Box( p+24, p+26),  Box( p+28, p+30),  Box( p+32, p+34)};
// 2 selected boxes as query; center and upper right
Box query[2] = { Box( p+16, p+18),  Box( p+32, p+34)};

// callback function object writing results to an output iterator
template <class OutputIterator>
struct Report {
    OutputIterator it;
    Report( OutputIterator i) : it(i) {} // store iterator in object
    // We write the id-number of box a to the output iterator assuming
    // that box b (the query box) is not interesting in the result.
    void operator()( const Box& a, const Box&) { *it++ = a.id(); }
};
template <class Iter> // helper function to create the function object
Report<Iter> report( Iter it) { return Report<Iter>(it); }

int main() {
    // run the intersection algorithm and store results in a vector
    std::vector<std::size_t> result;
    CGAL::box_intersection_d( boxes, boxes+9, query, query+2,
                              report( std::back_inserter( result)));
    // sort, check, and show result
    std::sort( result.begin(), result.end());
    std::size_t check1[13] = {0,1,2,3,4,4,5,5,6,7,7,8,8};
    assert(result.size() == 13 &&
   std::equal(check1,check1+13,result.begin()));
    std::copy( result.begin(), result.end(),
               std::ostream_iterator<std::size_t>( std::cout, " "));
    std::cout << std::endl;

    // run it again but for different cutoff value and half-open boxes
    result.clear();
    CGAL::box_intersection_d( boxes, boxes+9, query, query+2,
                              report( std::back_inserter( result)),
                              std::ptrdiff_t(1),
                              CGAL::Box_intersection_d::HALF_OPEN);
    // sort, check, and show result
    std::sort( result.begin(), result.end());
    std::size_t check2[2]  = {4,8};
    assert(result.size() == 2 && std::equal(check2, check2+2,
   result.begin()));
    std::copy( result.begin(), result.end(),
```

```
                std::ostream_iterator<std::size_t>( std::cout, " "));
    std::cout << std::endl;
    return 0;
}
```

File: examples/Box_intersection_d/box_grid.cpp

63.8 Runtime Performance

The implemented algorithm is described in [ZE02] as version two. Its performance depends on a *cutoff* parameter. When the size of both iterator ranges drops below the *cutoff* parameter the function switches from the streamed segment-tree algorithm to the two-way-scan algorithm, see [ZE02] for the details.

The streamed segment-tree algorithm needs $O(n\log^d(n)+k)$ worst-case running time and $O(n)$ space, where n is the number of boxes in both input sequences, d the (constant) dimension of the boxes, and k the output complexity, i.e., the number of pairwise intersections of the boxes. The two-way-scan algorithm needs $O(n\log(n)+l)$ worst-case running time and $O(n)$ space, where l is the number of pairwise overlapping intervals in one dimensions (the dimension where the algorithm is used instead of the segment tree). Note that l is not necessarily related to k and using the two-way-scan algorithm is a heuristic.

Unfortunately, we have no general method to automatically determine an optimal cutoff parameter, since it depends on the used hardware, the runtime ratio between callback runtime and segment-tree runtime, and of course the number of boxes to be checked and their distribution. In cases where the callback runtime is dominant, it may be best to make the threshold parameter small. Otherwise a $cutoff = \sqrt{n}$ can lead to acceptable results. For well distributed boxes the original paper [ZE02] gives optimal cutoffs in the thousands. Anyway, for optimal runtime some experiments to compare different cutoff parameters are recommended.

To demonstrate that box intersection can be done quite fast, different box sequences are intersected in the range between 4 and 800000 boxes total. We use three-dimensional default boxes of closed topology with *float* coordinates and without additional data fields. The algorithm works directly on the boxes, not on pointer to boxes. Each box intersection is reported to an empty dummy callback.

For each box set, a near-optimal cutoff parameter is determined using an adaptive approximation. The runtime required for streaming is compared against usual scanning. Results on a Xeon 2.4GHz with 4GB main memory can be seen in Figure 63.1. For a small number of boxes, pure scanning is still faster than streaming with optimal cutoff, which would just delegate the box sets to the scanning algorithm. As there are more and more boxes, the overhead becomes less important.

63.9 Example Using a Custom Box Implementation

The example in the previous Section 63.7 uses an array to provide the coordinates and then creates another array for the boxes. In the following example we write our own box class *Box* that we can initialize directly with the four coordinates and create the array of boxes directly. We also omit the explicitly stored *id*-number and use the address of the box itself as *id*-number. This works only if the boxes do not change their position, i.e., we work with pointers to the boxes in the intersection algorithm.

We follow with our own box class *Box* the *BoxIntersectionBox_d* concept, which allows us to reuse the default traits implementation, i.e., we can use the same default function call to compute all intersections. See the

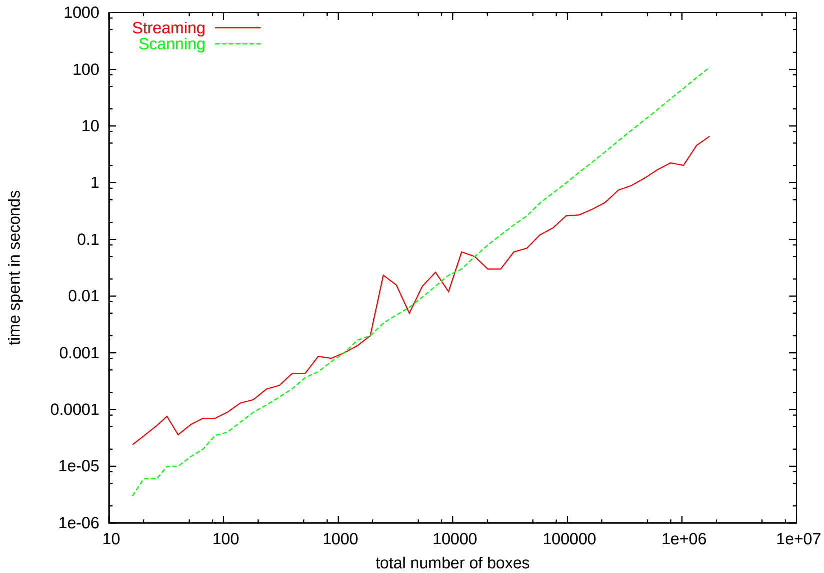


Figure 63.1: Runtime comparison between the scanning and the streaming algorithm.

example in the next section for a self-written traits class. So, in principle, the remainder of the example stays the same and we omit the part from the previous example for brevity that illustrates the half-open box topology.

The requirements for the box implementation are best studied on page 3714 in the Reference Manual. In a nutshell, we have to define the type *NT* for the box coordinates and the type *ID* for the *id*-number. Member functions give access to the coordinates and the *id*-number. A static member function returns the dimension.

```
#include <CGAL/box_intersection_d.h>
#include <vector>
#include <algorithm>
#include <iterator>
#include <cassert>

struct Box {
    typedef int             NT;
    typedef std::ptrdiff_t ID;
    int lo[2], hi[2];
    Box( int lo0, int lo1, int hi0, int hi1) { lo[0]=lo0; lo[1]=lo1;
   hi[0]=hi0; hi[1]=hi1;}
    static int dimension() { return 2; }
    int min_coord(int dim) const { return lo[dim]; }
    int max_coord(int dim) const { return hi[dim]; }
    // id-function using address of current box,
    // requires to work with pointers to boxes later
    std::ptrdiff_t id() const { return (std::ptrdiff_t)(this); }
};

// 9 boxes of a grid
Box boxes[9] = { Box( 0,0,1,1),  Box( 1,0,2,1),  Box( 2,0,3,1), // low
                 Box( 0,1,1,2),  Box( 1,1,2,2),  Box( 2,1,3,2), // middle
                 Box( 0,2,1,3),  Box( 1,2,2,3),  Box( 2,2,3,3)};// upper
// 2 selected boxes as query; center and upper right
```

```
Box query[2] = { Box( 1,1,2,2),  Box( 2,2,3,3)};

// With the special id-function we need to work on box pointers
Box* b_ptr[9] = { boxes,   boxes+1, boxes+2, boxes+3, boxes+4, boxes+5,
                  boxes+6, boxes+7, boxes+8};
Box* q_ptr[2] = { query,   query+1};

// callback function object writing results to an output iterator
template <class OutputIterator>
struct Report {
    OutputIterator it;
    Report( OutputIterator i) : it(i) {} // store iterator in object
    // We write the position with respect to 'boxes' to the output
   iterator
    // assuming that box b (the query box) is not interesting in the
   result.
    void operator()( const Box* a, const Box*) {
        *it++ = ( reinterpret_cast<Box*>(a->id()) - boxes);
    }
};
template <class Iter> // helper function to create the function object
Report<Iter> report( Iter it) { return Report<Iter>(it); }

int main() {
    // run the intersection algorithm and store results in a vector
    std::vector<std::size_t> result;
    CGAL::box_intersection_d( b_ptr, b_ptr+9, q_ptr, q_ptr+2,
                              report( std::back_inserter( result)),
                              std::ptrdiff_t(0));
    // sort and check result
    std::sort( result.begin(), result.end());
    std::size_t chk[13] = {0,1,2,3,4,4,5,5,6,7,7,8,8};
    assert( result.size()==13 && std::equal(chk,chk+13,result.begin()));
    return 0;
}
```

File: examples/Box_intersection_d/custom_box_grid.cpp

63.10 Example for Point Proximity Search with a Custom Traits Class

Given a set of 3D points, we want to find all pairs of points that are less than a certain distance apart. We use the box intersection algorithm to find good candidates, namely those that are less than this specified distance apart in the L_∞ norm, which is a good approximation of the Euclidean norm.

We use an unusual representation for the box, namely pointers to the 3D points themselves. We implement a special box traits class that interprets the point as a box of the dimensions $[-eps, +eps]^3$ centered at this point. The value for *eps* is half the specified distance from above, i.e., points are reported if their distance is smaller than *2*eps*.

The requirements for the box traits class are best studied on page 3715 in the Reference Manual. In a nutshell, we have to define the type *NT* for the box coordinates, the type *ID* for the *id*-number, and the type *Box_parameter* similar to the box handle, here *Point_3** since we work with the pointers. All member functions in the traits class are static. Two functions give access to the max and min coordinates that we compute from the point coordinates plus or minus the *eps* value, respectively. For the *id*-number function the address of the point itself is sufficient, since the points stay stable. Another function returns the dimension.

The *report* callback function computes than the Euclidean distance and prints a message for points that are close enough.

Note that we need to reserve sufficient space in the *points* vector to avoid reallocations while we create the *points* vector and the *boxes* vector in parallel, since otherwise the *points* vector might reallocate and invalidate all pointers stored in the *boxes* so far.

```
#include <CGAL/Simple_cartesian.h>
#include <CGAL/box_intersection_d.h>
#include <CGAL/point_generators_3.h>
#include <CGAL/algorithm.h>
#include <vector>
#include <algorithm>
#include <iterator>
#include <cmath>

typedef CGAL::Simple_cartesian<float>             Kernel;
typedef Kernel::Point_3                           Point_3;
typedef CGAL::Random_points_on_sphere_3<Point_3>  Points_on_sphere;

std::vector<Point_3>  points;
std::vector<Point_3*> boxes;     // boxes are just pointers to points
const float           eps = 0.1f; // finds point pairs of distance <
    2*eps

// Boxes are just pointers to 3d points. The traits class adds the
// +- eps size to each interval around the point, effectively building
// on the fly a box of size 2*eps centered at the point.
struct Traits {
    typedef float          NT;
    typedef Point_3*       Box_parameter;
    typedef std::ptrdiff_t ID;

    static int   dimension() { return 3; }
    static float coord( Box_parameter b, int d) {
        return (d == 0) ? b->x() : ((d == 1) ? b->y() : b->z());
    }
    static float min_coord( Box_parameter b, int d) { return
    coord(b,d)-eps;}
    static float max_coord( Box_parameter b, int d) { return
    coord(b,d)+eps;}
    // id-function using address of current box,
    // requires to work with pointers to boxes later
    static std::ptrdiff_t id(Box_parameter b) { return
    (std::ptrdiff_t)(b); }
};

// callback function reports pairs in close proximity
void report( const Point_3* a, const Point_3* b) {
    float dist = std::sqrt( CGAL::squared_distance( *a, *b));
```

```
    if ( dist < 2*eps) {
        std::cout << "Point " << (a - &(points.front())) << " and Point "
                  << (b - &(points.front())) << " have distance " << dist
                  << "." << std::endl;
    }
}

int main() {
    // create some random points on the sphere of radius 1.0
    Points_on_sphere generator( 1.0);
    points.reserve( 50);
    for ( int i = 0; i != 50; ++i) {
        points.push_back( *generator++);
        boxes.push_back( & points.back());
    }

    // run the intersection algorithm and report proximity pairs
    CGAL::box_self_intersection_d( boxes.begin(), boxes.end(),
                                   report, Traits());
    return 0;
}
```

File: examples/Box_intersection_d/proximity_custom_box_traits.cpp

63.11 Design and Implementation History

Lutz Kettner and Andreas Meyer implemented the algorithms starting from the publication [ZE02]. We had access to the original C implementation of Afra Zomorodian, which helped clarifying some questions, and we are grateful to the help of Afra Zomorodian in answering our questions during his visit. We thank Steve Robbins for an excellent review for this package. Steve Robbins provided an independent and earlier implementation of this algorithm, however, we learned too late about this implementation.

Intersecting Sequences of dD Iso-oriented Boxes Reference Manual

Lutz Kettner, Andreas Meyer, and Afra Zomorodian

We provide an efficient algorithm [ZE02] for finding all intersecting[2] pairs for large numbers of iso-oriented boxes, i.e., typically these will be bounding boxes of more complicated geometries. For comparison and as a base case for other methods, we also offer the simple all-pairs test as a generic function.

63.12 Classified Reference Pages

Concepts

Classes

Functions

[2]Boxes represent volumes or point-sets. So, *intersection* means intersection of the point-set enclosed by the box and not only intersection of the boundary, of course.

63.13 Alphabetical List of Reference Pages

CGAL::box_intersection_d

Definition

The function *box_intersection_d* computes the pairwise intersecting boxes between two sequences of iso-oriented boxes in arbitrary dimension. The sequences of boxes are given with two random-access iterator ranges and will be reordered in the course of the algorithm. For each intersecting pair of boxes a *callback* function object is called with the two intersecting boxes as argument; the first argument is a box from the first sequence, the second argument a box from the second sequence. The performance of the algorithm can be tuned with a *cutoff* parameter, see the implementation section below for more details.

The algorithm reorders the boxes in the course of the algorithm. Now, depending on the size of a box it can be faster to copy the boxes, or to work with pointers to boxes and copy only pointers. We offer automatic support for both options. To simplify the description, let us call the *value_type* of the iterators *box handle*. The *box handle* can either be our box type itself or a pointer (or const pointer) to the box type.

A d-dimensional iso-oriented box is defined as the Cartesian product of d intervals. We call the box *half-open* if the d intervals $\{[lo_i,hi_i)\,|\,0 \leq i < d\}$ are half-open intervals, and we call the box *closed* if the d intervals $\{[lo_i,hi_i]\,|\,0 \leq i < d\}$ are closed intervals. Note that closed boxes support zero-width boxes and they can intersect at their boundaries, while non-empty half-open boxes always have a positive volume and they only intersect iff their interiors overlap. The distinction between closed or half-open boxes does not require a different representation of boxes, just a different interpretation when comparing boxes, which is selected with the *topology* parameter and its two values, *CGAL::Box_intersection_d::HALF_OPEN* and *CGAL::Box_intersection_d::CLOSED*.

In addition, a box has an unique *id*-number. It is used to order boxes consistently in each dimension even if boxes have identical coordinates. In consequence, the algorithm guarantees that a pair of intersecting boxes is reported only once. Boxes with equal *id*-number are not reported since they obviously intersect trivially.

The algorithm uses a traits class of the *BoxIntersectionTraits_d* concept to access the boxes. A default traits class is provided that assumes that the box type is a model of the *BoxIntersectionBox_d* concept and that the box handle, i.e., the iterators value type, is identical to the box type or a pointer to the box type.

An important special application of this algorithm is the test for self-intersections where the second box sequence is an identical copy of the first sequence including the preserved *id*-number. Note that this implies that the address of the box is not sufficient for the *id*-number if boxes are copied by value. To ease the use of this special case we offer a simplified version of this function with one iterator range only, which then creates internally the second copy of the boxes, under the name *CGAL::box_self_intersection_d*.

In the general case, we distinguish between the bipartite case (the boxes are from different sequences) and the complete case (the boxes are from the same sequence, i.e., the self intersection case). The default is the bipartite case, since the complete case is typically handled with the simplified function call mentioned above. However, the general function call offers the *setting* parameter with the values *CGAL::Box_intersection_d::COMPLETE* and *CGAL::Box_intersection_d::BIPARTITE*.

#include <CGAL/box_intersection_d.h>

template< class RandomAccessIterator1, class RandomAccessIterator2, class Callback >
void box_intersection_d(RandomAccessIterator1 begin1,

RandomAccessIterator1 end1,
RandomAccessIterator2 begin2,
RandomAccessIterator2 end2,
Callback callback,
std::ptrdiff_t cutoff = 10,
Box_intersection_d::Topology topology = Box_intersection_d::CLOSED,
Box_intersection_d::Setting setting = Box_intersection_d::BIPARTITE)

Invocation of box intersection with default box traits *CGAL::Box_intersection_d::Box_traits_d<Box_handle>*, where *Box_handle* corresponds to the iterator value type of *RandomAccessIterator1*.

template< class RandomAccessIterator1, class RandomAccessIterator2, class Callback, class BoxTraits >
void *box_intersection_d(RandomAccessIterator1 begin1,*
RandomAccessIterator1 end1,
RandomAccessIterator2 begin2,
RandomAccessIterator2 end2,
Callback callback,
BoxTraits box_traits,
std::ptrdiff_t cutoff = 10,
Box_intersection_d::Topology topology = Box_intersection_d::CLOSED,
Box_intersection_d::Setting setting = Box_intersection_d::BIPARTITE)

Invocation with custom box traits.

Requirements

- *RandomAccessIterator1*, and *...2*, must be mutable random-access iterators and both value types must be the same. We call this value type *Box_handle* in the following.
- *Callback* must be of the *BinaryFunction* concept. The *Box_handle* must be convertible to both argument types. The return type is not used and can be *void*.
- The *Box_handle* must be a model of the *Assignable* concept.
- In addition, if the default box traits is used the *Box_handle* must be a class type *T* or a pointer to a class type *T*, where *T* must be a model of the *BoxIntersectionBox_d* concept. In both cases, the default box traits specializes to a suitable implementation.
- *BoxTraits* must be of the *BoxIntersectionTraits_d* concept.

See Also

CGAL::box_self_intersection_d .... page 3709
CGAL::box_intersection_all_pairs_d .... page 3707

CGAL::Box_intersection_d::Box_traits_d<BoxHandle> .... page 3720
BoxIntersectionBox_d .... page 3714
BoxIntersectionTraits_d .... page 3715

Implementation

The implemented algorithm is described in [ZE02] as version two. Its performance depends on a *cutoff* parameter. When the size of both iterator ranges drops below the *cutoff* parameter the function switches from the streamed segment-tree algorithm to the two-way-scan algorithm, see [ZE02] for the details.

The streamed segment-tree algorithm needs $O(n\log^d(n)+k)$ worst-case running time and $O(n)$ space, where n is the number of boxes in both input sequences, d the (constant) dimension of the boxes, and k the output complexity, i.e., the number of pairwise intersections of the boxes. The two-way-scan algorithm needs $O(n\log(n)+l)$ worst-case running time and $O(n)$ space, where l is the number of pairwise overlapping intervals in one dimensions (the dimension where the algorithm is used instead of the segment tree). Note that l is not necessarily related to k and using the two-way-scan algorithm is a heuristic.

Unfortunately, we have no general method to automatically determine an optimal cutoff parameter, since it depends on the used hardware, the runtime ratio between callback runtime and segment-tree runtime, and of course the number of boxes to be checked and their distribution. In cases where the callback runtime is dominant, it may be best to make the threshold parameter small. Otherwise a $cutoff = \sqrt{n}$ can lead to acceptable results. For well distributed boxes the original paper [ZE02] gives optimal cutoffs in the thousands. Anyway, for optimal runtime some experiments to compare different cutoff parameters are recommended. See also Section 63.8.

Example

The box implementation provided with *CGAL::Box_intersection_d::Box_d<double,2>* has a special constructor for the CGAL bounding box type *CGAL::Bbox_2* (and similar for dimension 3). We use this in the example to create 3×3 *boxes* in a grid layout. Additionally we pick the center box and the box in the upper-right corner as our second box sequence *query*.

The default policy of the box type implements the *id*-number with an explicit counter in the boxes, which is the default choice since it always works. We use the *id*-number in our callback function to report the result of the intersection algorithm call. The result will be that the first *query* box intersects all nine *boxes* and the second *query* box intersects the four boxes in the upper-right quadrant.

```
#include <CGAL/box_intersection_d.h>
#include <CGAL/Bbox_2.h>
#include <iostream>

typedef CGAL::Box_intersection_d::Box_d<double,2> Box;
typedef CGAL::Bbox_2                              Bbox;
                                                     // 9 boxes of a grid
Box boxes[9] = { Bbox( 0,0,1,1), Bbox( 1,0,2,1), Bbox( 2,0,3,1), // low
                 Bbox( 0,1,1,2), Bbox( 1,1,2,2), Bbox( 2,1,3,2), //
    middle
                 Bbox( 0,2,1,3), Bbox( 1,2,2,3), Bbox( 2,2,3,3)};// upper
// 2 selected boxes as query; center and upper right
Box query[2] = { Bbox( 1,1,2,2), Bbox( 2,2,3,3)};

void callback( const Box& a, const Box& b ) {
    std::cout << "box " << a.id() << " intersects box " << b.id() <<
   std::endl;
}
int main() {
    CGAL::box_intersection_d( boxes, boxes+9, query, query+2, callback);
    return 0;
}
```

File: examples/Box_intersection_d/minimal.cpp

CGAL::box_intersection_all_pairs_d

Definition

The function *box_intersection_all_pairs_d* computes the pairwise intersecting boxes between two sequences of iso-oriented boxes in arbitrary dimension. It does so by comparing all possible pairs of boxes and is thus inferior to the fast *CGAL::box_intersection_d* algorithm on page 3703.

The sequences of boxes are given with two forward iterator ranges. The sequences are not modified. For each intersecting pair of boxes a *callback* function object is called with the two intersecting boxes as argument; the first argument is a box from the first sequence, the second argument a box from the second sequence.

The algorithm is interface compatible with the *CGAL::box_intersection_d* function. Similarly, we call the *value_type* of the iterators the *box handle*, which is either our box type or a pointer type to our box type.

A d-dimensional iso-oriented box is defined as the Cartesian product of d intervals. We call the box *half-open* if the d intervals $\{[lo_i,hi_i)\,|\,0 \leq i < d\}$ are half-open intervals, and we call the box *closed* if the d intervals $\{[lo_i,hi_i]\,|\,0 \leq i < d\}$ are closed intervals. Note that closed boxes support zero-width boxes and they can intersect at their boundaries, while non-empty half-open boxes always have a positive volume and they only intersect iff their interiors overlap. The distinction between closed or half-open boxes does not require a different representation of boxes, just a different interpretation when comparing boxes, which is selected with the *topology* parameter and its two values, *CGAL::Box_intersection_d::HALF_OPEN* and *CGAL::Box_intersection_d::CLOSED*.

In addition, a box has an unique *id*-number. Boxes with equal *id*-number are not reported since they obviously intersect trivially.

The algorithm uses a traits class of the *BoxIntersectionTraits_d* concept to access the boxes. A default traits class is provided that assumes that the box type is a model of the *BoxIntersectionBox_d* concept and that the box handle, i.e., the iterators value type, is identical to the box type or a pointer to the box type.

An important special application of this algorithm is the test for self-intersections where the second box sequence is an identical copy of the first sequence including the preserved *id*-number. We offer a specialized implementation *CGAL::box_self_intersection_all_pairs* for this application.

#include <CGAL/box_intersection_d.h>

```
template< class ForwardIterator1, class ForwardIterator2, class Callback >
void    box_intersection_all_pairs_d( ForwardIterator1 begin1,
                                      ForwardIterator1 end1,
                                      ForwardIterator2 begin2,
                                      ForwardIterator2 end2,
                                      Callback callback,
                                      Box_intersection_d::Topology topology = Box_intersection_d::CLOSED)
```

Invocation of box intersection with default box traits *CGAL::Box_intersection_d::Box_traits_d<Box_handle>*, where *Box_handle* corresponds to the iterator value type of *ForwardIterator1*.

```
template< class ForwardIterator1, class ForwardIterator2, class Callback, class BoxTraits >
void    box_intersection_all_pairs_d( ForwardIterator1 begin1,
```

ForwardIterator1 end1,
ForwardIterator2 begin2,
ForwardIterator2 end2,
Callback callback,
BoxTraits box_traits,
Box_intersection_d::Topology topology = Box_intersection_d::CLOSED)

Invocation with custom box traits.

Requirements

- *ForwardIterator1*, and ...*2*, must be forward iterators and both value types must be the same. We call this value type *Box_handle* in the following.
- *Callback* must be of the *BinaryFunction* concept. The *Box_handle* must be convertible to both argument types. The return type is not used and can be *void*.
- The *Box_handle* must be a model of the *Assignable* concept.
- In addition, if the default box traits is used the *Box_handle* must be a class type T or a pointer to a class type T, where T must be a model of the *BoxIntersectionBox_d* concept. In both cases, the default box traits specializes to a suitable implementation.
- *BoxTraits* must be of the *BoxIntersectionTraits_d* concept.

See Also

CGAL::box_intersection_d .... page 3703
CGAL::box_self_intersection_d .... page 3709
CGAL::box_self_intersection_all_pairs_d .... page 3712

CGAL::Box_intersection_d::Box_traits_d<BoxHandle> .... page 3720
BoxIntersectionBox_d .... page 3714
BoxIntersectionTraits_d .... page 3715

Implementation

The algorithm is trivially testing all pairs and runs therefore in time $O(nm)$ where n is the size of the first sequence and m is the size of the second sequence.

CGAL::box_self_intersection_d

Function

Definition

The function *box_self_intersection_d* computes the pairwise intersecting boxes in a sequence of iso-oriented boxes in arbitrary dimension. The sequence of boxes is given with as a random-access iterator range and will be reordered in the course of the algorithm. For each intersecting pair of boxes a *callback* function object is called with the two intersecting boxes as argument; the first argument is a box from the sequence, the second argument is a copy of a box from the sequence. The performance of the algorithm can be tuned with a *cutoff* parameter, see the implementation section of the *CGAL::box_intersection_d* function on page 3703.

The algorithm creates a second copy of the boxes and reorders the boxes in the course of the algorithm. Now, depending on the size of a box it can be faster to copy the boxes, or to work with pointers to boxes and copy only pointers. We offer automatic support for both options. To simplify the description, let us call the *value_type* of the iterators *box handle*. The *box handle* can either be our box type itself or a pointer (or const pointer) to the box type.

A d-dimensional iso-oriented box is defined as the Cartesian product of d intervals. We call the box *half-open* if the d intervals $\{[lo_i, hi_i) \,|\, 0 \leq i < d\}$ are half-open intervals, and we call the box *closed* if the d intervals $\{[lo_i, hi_i] \,|\, 0 \leq i < d\}$ are closed intervals. Note that closed boxes support zero-width boxes and they can intersect at their boundaries, while non-empty half-open boxes always have a positive volume and they only intersect iff their interiors overlap. The distinction between closed or half-open boxes does not require a different representation of boxes, just a different interpretation when comparing boxes, which is selected with the *topology* parameter and its two values, *CGAL::Box_intersection_d::HALF_OPEN* and *CGAL::Box_intersection_d::CLOSED*.

In addition, a box has an unique *id*-number. It is used to order boxes consistently in each dimension even if boxes have identical coordinates. In consequence, the algorithm guarantees that a pair of intersecting boxes is reported only once. This self-intersection function creates internally a second copy of the box sequence. The copying has to preserve the *id*-number of boxes. Note that this implies that the address of the box is not sufficient for the *id*-number if boxes are copied by value. Boxes of equal *id*-number are not reported as intersecting pairs since they are always intersecting trivially.

The algorithm uses a traits class of the *BoxIntersectionTraits_d* concept to access the boxes. A default traits class is provided that assumes that the box type is a model of the *BoxIntersectionBox_d* concept and that the box handle, i.e., the iterators value type, is identical to the box type or a pointer to the box type.

#include <CGAL/box_intersection_d.h>

```
template< class RandomAccessIterator, class Callback >
void    box_self_intersection_d( RandomAccessIterator begin,
                                 RandomAccessIterator end,
                                 Callback callback,
                                 std::ptrdiff_t cutoff = 10,
                                 Box_intersection_d::Topology topology = Box_intersection_d::CLOSED)
```

Invocation of box intersection with default box traits *CGAL::Box_intersection_d::Box_traits_d<Box_handle>*, where *Box_handle* corresponds to the iterator value type of *RandomAccessIterator*.

```
template< class RandomAccessIterator, class Callback, class BoxTraits >
void    box_self_intersection_d( RandomAccessIterator begin,
```

RandomAccessIterator end,
Callback callback,
BoxTraits box_traits,
std::ptrdiff_t cutoff = 10,
Box_intersection_d::Topology topology = Box_intersection_d::CLOSED)

Invocation with custom box traits.

Requirements

- *RandomAccessIterator* must be a mutable random-access iterator. We call its value type *Box_handle* in the following.
- *Callback* must be of the *BinaryFunction* concept. The *Box_handle* must be convertible to both argument types. The return type is not used and can be *void*.
- The *Box_handle* must be a model of the *Assignable* concept.
- In addition, if the default box traits is used the *Box_handle* must be a class type *T* or a pointer to a class type *T*, where *T* must be a model of the *BoxIntersectionBox_d* concept. In both cases, the default box traits specializes to a suitable implementation.
- *BoxTraits* must be of the *BoxIntersectionTraits_d* concept.

See Also

CGAL::box_intersection_d . . . . page 3703
CGAL::box_self_intersection_all_pairs_d . . . . page 3712

CGAL::Box_intersection_d::Box_traits_d<BoxHandle> . . . . page 3720
BoxIntersectionBox_d . . . . page 3714
BoxIntersectionTraits_d . . . . page 3715

Implementation

See the implementation section of the *CGAL::box_intersection_d* function on page 3703.

Example

The box implementation provided with *CGAL::Box_intersection_d::Box_d<double,2>* has a special constructor for the CGAL bounding box type *CGAL::Bbox_2* (and similar for dimension 3). We use this in the example to create 3×3 *boxes* in a grid layout.

The default policy of the box type implements the *id*-number with an explicit counter in the boxes, which is the default choice since it always works. We use the *id*-number in our callback function to report the result of the intersection algorithm call. The result will be 20 pairwise intersections, but the order in which they are reported is non-intuitive.

```
#include <CGAL/box_intersection_d.h>
#include <CGAL/Bbox_2.h>
#include <iostream>

typedef CGAL::Box_intersection_d::Box_d<double,2> Box;
typedef CGAL::Bbox_2                              Bbox;

// 9 boxes of a grid
Box boxes[9] = { Bbox( 0,0,1,1), Bbox( 1,0,2,1), Bbox( 2,0,3,1), // low
                 Bbox( 0,1,1,2), Bbox( 1,1,2,2), Bbox( 2,1,3,2), //
   middle
                 Bbox( 0,2,1,3), Bbox( 1,2,2,3), Bbox( 2,2,3,3)};// upper

void callback( const Box& a, const Box& b ) {
    std::cout << "box " << a.id() << " intersects box " << b.id() <<
   std::endl;
}

int main() {
    CGAL::box_self_intersection_d( boxes, boxes+9, callback);
    return 0;
}
```

File: examples/Box_intersection_d/minimal_self.cpp

Function

CGAL::box_self_intersection_all_pairs_d

Definition

The function *box_self_intersection_all_pairs_d* computes the pairwise intersecting boxes in a sequence of iso-oriented boxes in arbitrary dimension. It does so by comparing all possible pairs of boxes and is thus inferior to the fast *CGAL::box_self_intersection_d* algorithm on page 3709.

The sequence of boxes is given with a forward iterator range. The sequences are not modified. For each intersecting pair of boxes a *callback* function object is called with the two intersecting boxes as argument.

The algorithm is interface compatible with the *CGAL::box_self_intersection_d* function. Similarly, we call the *value_type* of the iterators the *box handle*, which is either our box type or a pointer type to our box type.

A d-dimensional iso-oriented box is defined as the Cartesian product of d intervals. We call the box *half-open* if the d intervals $\{[lo_i, hi_i) \,|\, 0 \leq i < d\}$ are half-open intervals, and we call the box *closed* if the d intervals $\{[lo_i, hi_i] \,|\, 0 \leq i < d\}$ are closed intervals. Note that closed boxes support zero-width boxes and they can intersect at their boundaries, while non-empty half-open boxes always have a positive volume and they only intersect iff their interiors overlap. The distinction between closed or half-open boxes does not require a different representation of boxes, just a different interpretation when comparing boxes, which is selected with the *topology* parameter and its two values, *CGAL::Box_intersection_d::HALF_OPEN* and *CGAL::Box_intersection_d::CLOSED*.

The algorithm uses a traits class of the *BoxIntersectionTraits_d* concept to access the boxes. A default traits class is provided that assumes that the box type is a model of the *BoxIntersectionBox_d* concept and that the box handle, i.e., the iterators value type, is identical to the box type or a pointer to the box type.

#include <CGAL/box_intersection_d.h>

```
template< class ForwardIterator, class Callback >
void    box_self_intersection_all_pairs_d(
            ForwardIterator begin,
            ForwardIterator end,
            Callback callback,
            Box_intersection_d::Topology topology = Box_intersection_d::CLOSED)
```

Invocation of box intersection with default box traits *CGAL::Box_intersection_d::Box_traits_d<Box_handle>*, where *Box_handle* corresponds to the iterator value type of *ForwardIterator*.

```
template< class ForwardIterator, class Callback, class BoxTraits >
void    box_self_intersection_all_pairs_d(
            ForwardIterator begin,
            ForwardIterator end,
            Callback callback,
            BoxTraits box_traits,
            Box_intersection_d::Topology topology = Box_intersection_d::CLOSED)
```

Invocation with custom box traits.

Requirements

- *ForwardIterator* must be a forward iterator. We call its value type *Box_handle* in the following.
- *Callback* must be of the *BinaryFunction* concept. The *Box_handle* must be convertible to both argument types. The return type is not used and can be *void*.
- The *Box_handle* must be a model of the *Assignable* concept.
- In addition, if the default box traits is used the *Box_handle* must be a class type *T* or a pointer to a class type *T*, where *T* must be a model of the *BoxIntersectionBox_d* concept. In both cases, the default box traits specializes to a suitable implementation.
- *BoxTraits* must be of the *BoxIntersectionTraits_d* concept.

See Also

CGAL::box_intersection_d . . . . page 3703
CGAL::box_self_intersection_d . . . . page 3709
CGAL::box_intersection_all_pairs_d . . . . page 3707

CGAL::Box_intersection_d::Box_traits_d<BoxHandle> . . . . page 3720
BoxIntersectionBox_d . . . . page 3714
BoxIntersectionTraits_d . . . . page 3715

Implementation

The algorithm is trivially testing all pairs and runs therefore in time $O(n^2)$ where n is the size of the input sequence. This algorithm does not use the id-number of the boxes.

Concept

BoxIntersectionBox_d

Definition

The BoxIntersectionBox_d concept is used in the context of the intersection algorithms for sequences of iso-oriented boxes. These algorithms come with a default traits class that assumes that the boxes are a model of this BoxIntersectionBox_d concept. This concept defines the access functions to the dimension, the *id*-number, and the boundaries of the box.

Refines

Assignable.

Has Models

CGAL::Box_intersection_d::Box_d<NT,int D,IdPolicy> . . . . . page 3717
CGAL::Box_intersection_d::Box_with_handle_d<NT, int D, Handle, IdPolicy> . . . . . page 3721

Types

BoxIntersectionBox_d:: NT	number type to represent the box boundaries. Allowed are the built-in types *int*, *unsigned int*, *float*, and *double*.
BoxIntersectionBox_d:: ID	type for the box *id*-number, must be a model of the *LessThanComparable* concept.

Access Functions

int *BoxIntersectionBox_d::dimension()*

returns the dimension of the box.

ID	*box.id() const*	returns the unique *id*-number for the *box*.
NT	*box.min_coord(int d) const*	returns the lower boundary in dimension d, $0 \leq d < dimension()$.
NT	*box.max_coord(int d) const*	returns the upper boundary in dimension d, $0 \leq d < dimension()$.

See Also

CGAL::box_intersection_d . . . . . page 3703
CGAL::box_self_intersection_d . . . . . page 3709
CGAL::box_intersection_all_pairs_d . . . . . page 3707
CGAL::box_self_intersection_all_pairs_d . . . . . page 3712

CGAL::Box_intersection_d::Box_traits_d<BoxHandle> . . . . . page 3720
BoxIntersectionTraits_d . . . . . page 3715

BoxIntersectionTraits_d

Concept

Definition

The BoxIntersectionTraits_d concept is used for the intersection algorithms for sequences of iso-oriented boxes. This concept defines the access functions to the dimension, the *id*-number, and the boundaries of the boxes manipulated in these algorithms.

Refines

Assignable.

Has Models

CGAL::Box_intersection_d::Box_traits_d<BoxHandle> . . . . page 3720

Types

BoxIntersectionTraits_d:: Box_parameter

type used for passing box parameters in the functions below. Since we support in our algorithms passing the boxes by value as well as passing them as pointers, this type can be either *const B*`&`, *B**, or *const B** respectively, where *B* is the actual box type. The difference to the box handle type lies in the first case where the box handle would be *B* where this type is *const B*`&`.

BoxIntersectionTraits_d:: NT — number type to represent the box boundaries. Allowed are the built-in types *int*, *unsigned int*, *float*, and *double*.

BoxIntersectionTraits_d:: ID — type for the *id*-number, model of the *LessThanComparable* concept.

Access Functions

int *BoxIntersectionTraits_d::dimension()* — returns the dimension of the box.

ID *BoxIntersectionTraits_d::id(Box_parameter box)* — returns the unique *id*-number for the *box*.

NT *BoxIntersectionTraits_d::min_coord(Box_parameter box, int d)*

returns the lower boundary of the *box* in dimension *d*, $0 \leq d < dimension()$.

NT *BoxIntersectionTraits_d::max_coord(Box_parameter box, int d)*

returns the upper boundary of the *box* in dimension *d*, $0 \leq d < dimension()$.

See Also

CGAL::box_intersection_d . . . page 3703
CGAL::box_self_intersection_d . . . page 3709
CGAL::box_intersection_all_pairs_d . . . page 3707
CGAL::box_self_intersection_all_pairs_d . . . page 3712

CGAL::Box_intersection_d::Box_d<NT,int D,IdPolicy>

Class

Definition

Box_d<NT,int D,IdPolicy> is a generic iso-oriented bounding box in dimension *D*. It provides in each dimension an interval with lower and upper endpoints represented with the number type *NT*. This class is designed to work smoothly with the algorithms for intersecting sequences of iso-oriented boxes. For degeneracy handling, the boxes need to provide a unique *id*-number. The policy parameter *IdPolicy* offers several choices. The template parameters have to comply with the following requirements:

- *NT*: number type for the box boundaries, needs to be a model of the *Assignable* and the *LessThanComparable* concept.
- *int D*: the dimension of the box.
- *IdPolicy*: specifies how the *id*-number will be provided. Can be one of the following types, where *ID_EXPLICIT* is the default for this parameter:
  - *ID_NONE*: no *id*-number is provided. Can be useful if *Box_d* is used as a base class for a different implementation of *id*-numbers than the ones provided here.
  - *ID_EXPLICIT*: the *id*-number is stored explicitly in the box and automatically created and assigned at construction time of the box. Note that copying a box (copy-constructor and assignment) does not create a new *id*-number but keeps the old one, which is the behavior needed by the *CGAL::box_self_intersection* algorithm. This is therefore the safe default implementation.
  - *ID_FROM_BOX_ADDRESS*: casts the address of the box into a *std::ptrdiff_t* to create the *id*-number. Works fine if the intersection algorithms work effectively with pointers to boxes, but not in the case where the algorithms work with box values, because the algorithms modify the order of the boxes, and the *CGAL::box_self_intersection* algorithm creates copies of the boxes that would not have identical *id*-numbers.

#include <CGAL/Box_intersection_d/Box_d.h>
and also automatically with
#include <CGAL/box_intersection_d.h>

Is Model for the Concepts

BoxIntersectionBox_d . . . . page 3714

Types

Box_d<NT,int D,IdPolicy>:: NT	number type to represent the box boundaries. Allowed are the built-in types *int*, *unsigned int*, *float*, and *double*.
typedef std::size_t ID;	type for the box *id*-number.

Creation

Box_d<NT,int D,IdPolicy> box; — Default constructor. No particular initialization.

Box_d<NT,int D,IdPolicy> box(bool complete); — initializes to the complete or the empty space. If empty, all interval starting(end) points will be set to positive(negative) infinity.

Box_d<NT,int D,IdPolicy> box(NT lo[D], NT hi[D]);

initializes the box intervals to [*lo[i]*,*hi[i]*], $0 \leq i < D$.
Precondition: *lo[i]* $<$ *hi[i]* for $0 \leq i < D$.

Box_d<NT,int D,IdPolicy> box(Bbox_2 bbox); — constructs from bbox
Requirement: $D = 2$ and *NT*≡*double*.

Box_d<NT,int D,IdPolicy> box(Bbox_3 bbox); — constructs from bbox
Requirement: $D = 3$ and *NT*≡*double*.

Modifiers

void	*box.init(bool complete = false)*	initializes to the complete or the empty space. If empty, all interval starting(end) points will be set to positive(negative) infinity.
void	*box.extend(NT point[D])*	extend *box* to contain the old *box* and *point*.

Access Functions

int	*Box_d::dimension()*	returns *D*, the dimension of the box.
std::size_t	*box.id()*	returns a unique box id, see the *IdPolicy* template parameter above for the different choices. *Requirement*: *IdPolicy*≠*ID_NONE*
NT	*box.min_coord(int d) const*	returns the lower boundary in dimension *d* *Precondition*: $0 \leq d < D$
NT	*box.max_coord(int d) const*	returns the upper boundary in dimension *d* *Precondition*: $0 \leq d < D$
Bbox_2	*box.bbox() const*	returns the bounding box *Requirement*: $D = 2$ and *NT*≡*double*
Bbox_3	*box.bbox() const*	returns the bounding box *Requirement*: $D = 3$ and *NT*≡*double*
void	*box.extend(NT p[N])*	extends *box* to the smallest box that additionally contains the point represented by coordinates in *p*.

void *box.extend(std::pair<NT,NT> p[N])*

extends *box* to the smallest box that additionally contains the point represented by coordinate intervals in *p*.

See Also

Class

CGAL::Box_intersection_d::Box_traits_d<BoxHandle>

Definition

This is the default traits class for the intersection algorithms for iso-oriented boxes. There are actually three versions depending on the type of *BoxHandle*; there is one if *BoxHandle* is a class type and there are two if *BoxHandle* is a pointer type, one for a mutable and one for a const pointer, respectively.

This class implements the mapping from its *BoxHandle* argument to the *Box_parameter* type required in the *BoxIntersectionTraits_d* concept. In particular in the case where *BoxHandle* is a class type *B*, it defines *Box_parameter* to be of type *const B*&, while for the other cases it just uses the pointer type.

- *BoxHandle*: either a class type *B*, a pointer *B**, or a const-pointer *const B**, where *B* is a model of the *BoxIntersectionBox_d* concept.

#include <CGAL/Box_intersection_d/Box_traits_d.h> and also automatically with
#include <CGAL/box_intersection_d.h>

Is Model for the Concepts

BoxIntersectionTraits_d . . . . page 3715

Creation

Box_traits_d<BoxHandle> traits; default constructor.

See Also

CGAL::box_intersection_d . . . . page 3703
CGAL::box_self_intersection_d . . . . page 3709
CGAL::box_intersection_all_pairs_d . . . . page 3707

BoxIntersectionBox_d . . . . page 3714
CGAL::Box_intersection_d::Box_d<NT,int D,IdPolicy> . . . . page 3717

CGAL::Box_intersection_d::Box_with_handle_d<NT, int D, Handle, IdPolicy>

Class

Definition

Box_with_handle_d<NT, int D, Handle, IdPolicy> is a generic iso-oriented bounding box in dimension *D* that stores additionally a handle to some underlying geometric object. It provides in each dimension an interval with lower and upper endpoints represented with the number type *NT*. This class is designed to work smoothly with the algorithms for intersecting sequences of iso-oriented boxes. For degeneracy handling, the boxes need to provide a unique *id*-number. The policy parameter *IdPolicy* offers several choices. The template parameters have to comply with the following requirements:

- *NT*: number type for the box boundaries, needs to be a model of the *Assignable* and the *LessThanComparable* concept.
- *int D*: the dimension of the box.
- *Handle* concept, e.g., a pointer, an iterator, or a circulator.
- *IdPolicy*: specifies how the *id*-number will be provided. Can be one of the following types, where *ID_EXPLICIT* is the default for this parameter:
  - *ID_NONE*: no *id*-number is provided. Can be useful to have this class as a base class for different implementations of *id*-numbers than the ones provided here.
  - *ID_EXPLICIT*: the *id*-number is stored explicitly in the box and automatically created and assigned at construction time of the box. Note that copying a box (copy-constructor and assignment) does not create a new *id*-number but keeps the old one, which is the behavior needed by the *CGAL::box_self_intersection* algorithm. This is therefore the safe default implementation.
  - *ID_FROM_BOX_ADDRESS*: casts the address of the box into a *std::ptrdiff_t* to create the *id*-number. Works fine if the intersection algorithms work effectively with pointers to boxes, but not in the case where the algorithms work with box values, because the algorithms modify the order of the boxes, and the *CGAL::box_self_intersection* algorithm creates copies of the boxes that would not have identical *id*-numbers.
  - *ID_FROM_HANDLE*: casts the address of the value of the handle into a *std::ptrdiff_t* to create the *id*-number. Works in many conceivable settings, e.g., it works with boxes copied by value or by pointer, and the self intersection test. It will not work if there is no one-to-one mapping between boxes and the geometry that is referred to with the handles, i.e., this *id*-number scheme fails if a geometric object creates several boxes with the same handle value. Note that this option is not available for the *CGAL::Box_intersection_d::Box_d* type that does not store a handle.

#include <CGAL/Box_intersection_d/Box_with_handle_d.h>
and also automatically with
#include <CGAL/box_intersection_d.h>

Is Model for the Concepts

BoxIntersectionBox_d . . . . page 3714

Types

Box_with_handle_d<NT, int D, Handle, IdPolicy>:: NT

number type to represent the box boundaries. Allowed are the built-in types *int*, *unsigned int*, *float*, and *double*.

typedef std::size_t ID; type for the box *id*-number.

Creation

Box_with_handle_d<NT, int D, Handle, IdPolicy> box;

Default constructor. No particular initialization.

Box_with_handle_d<NT, int D, Handle, IdPolicy> box(bool complete, Handle h);

initializes to the complete or the empty space. If empty, all interval starting(end) points will be set to positive(negative) infinity, sets handle to *h*.

Box_with_handle_d<NT, int D, Handle, IdPolicy> box(NT lo[D], NT hi[D], Handle h);

initializes the box intervals to [*lo[i]*,*hi[i]*], $0 \leq i < D$ and sets the handle to *h*.
Precondition: *lo[i]* < *hi[i]* for $0 \leq i < D$.

Box_with_handle_d<NT, int D, Handle, IdPolicy> box(Bbox_2 bbox, Handle h);

constructs from bbox and sets the handle to *h*, exists iff $D = 2$ and *NT*≡*double*.

Box_with_handle_d<NT, int D, Handle, IdPolicy> box(Bbox_3 bbox, Handle h);

constructs from bbox and sets the handle to *h*, exists iff $D = 3$ and *NT*≡*double*.

Modifiers

void	*box.init(bool complete = false)*	initializes to the complete or the empty space. If empty, all interval starting(end) points will be set to positive(negative) infinity.
void	*box.extend(NT point[D])*	extend *box* to contain the old *box* and *point*.

Access Functions

Handle	*box.handle() const*	returns the handle stored in *box*.
int	*Box_with_handle_d::dimension()*	returns D, the dimension of the box.
std::size_t	*box.id()*	returns a unique box id, see the *IdPolicy* template parameter above for the different choices. Does not exist if *ID_NONE* has been chosen for the *IdPolicy*.
NT	*box.min_coord(int d) const*	returns the lower boundary in dimension d, $0 \leq d < D$.
NT	*box.max_coord(int d) const*	returns the upper boundary in dimension d, $0 \leq d < D$.
Bbox_2	*box.bbox() const*	returns the bounding box iff $D = 2$ and $NT \equiv double$.
Bbox_3	*box.bbox() const*	returns the bounding box iff $D = 3$ and $NT \equiv double$.

See Also

CGAL::box_intersection_d . . . . page 3703
CGAL::box_self_intersection_d . . . . page 3709
CGAL::box_intersection_all_pairs_d . . . . page 3707
CGAL::box_self_intersection_all_pairs_d . . . . page 3712

CGAL::Box_intersection_d::Box_traits_d<BoxHandle> . . . . page 3720
BoxIntersectionTraits_d . . . . page 3715

Chapter 64

3D Fast Intersection and Distance Computation (AABB Tree)

Pierre Alliez, Stéphane Tayeb, Camille Wormser

Contents

64.1 Introduction

The AABB tree component offers a static data structure and algorithms to perform efficient intersection and distance queries against sets of finite 3D geometric objects. The set of geometric objects stored in the data structure can be queried for intersection detection, intersection computation and distance. The intersection queries can be of any type, provided that the corresponding intersection predicates and constructors are implemented in the traits class. The distance queries are limited to point queries. Examples of intersection queries include line objects (rays, lines, segments) against sets of triangles, or plane objects (planes, triangles) against sets of segments. An example of a distance query consists of finding the closest point from a point query to a set of triangles.

Note that this component is not suited to the problem of finding all intersecting pairs of objects. We refer to the component *Box_intersection_d* (Intersecting Sequences of dD Iso-oriented Boxes) which can find all intersecting pairs of iso-oriented boxes.

The AABB tree data structure takes as input an iterator range of geometric data, which is then converted into primitives. From these primitives a hierarchy of axis-aligned bounding boxes (AABBs) is constructed and used to speed up intersection and distance queries (see Figure 64.1). Each primitive gives access to both one input geometric object (so-called datum) and one reference id to this object. A typical example primitive wraps a 3D triangle as datum and a face handle of a polyhedral surface as id. Each intersection query can return the intersection objects (e.g., 3D points or segments for ray queries) as well the as id (here the face handle) of the intersected primitives. Similarly, each distance query can return the closest point from the point query as well as the id of the closest primitive.

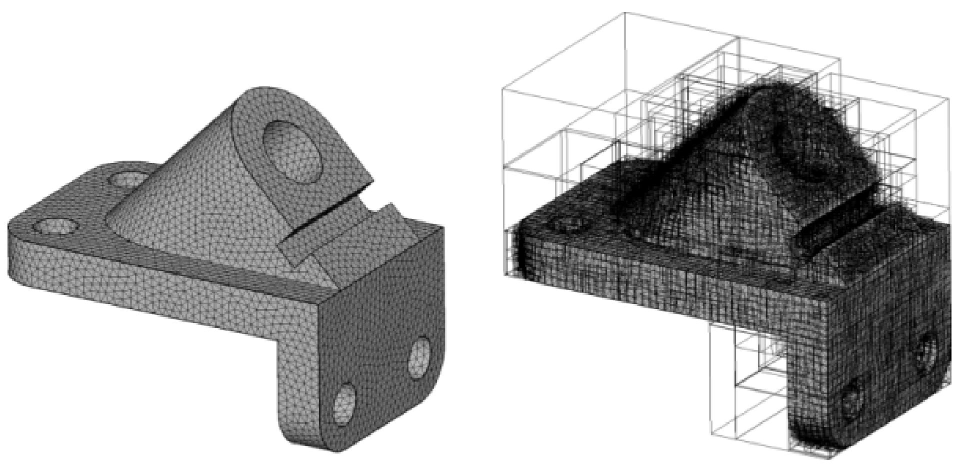


Figure 64.1: AABB tree. Left: surface triangle mesh of a mechanical part. Right: AABB tree constructed.

64.2 Interface

The main entry point to the component is the class *AABB_tree* which represents a static AABB tree constructed from an iterator range of geometric data. Once instantiated an AABB tree can be queried for intersection and distance queries.

Intersections. Assume for example that the tree contains triangle primitives. The tree can be queried for intersection against line objects (rays, segments or line) in various ways. We distinguish intersection *tests* which do not construct any intersection objects, from *intersections* which construct the intersection objects.

Tests:

- Function *do_intersect* tests if the input primitives are intersected by the query. This function is fast as it involves only predicates and stops after the first encountered intersection.
- Function *number_of_intersected_primitives* counts all intersected primitives.
- Function *all_intersected_primitives* enumerates all intersected primitives ids without constructing the corresponding intersection objects.
- Function *any_intersected_primitive* returns the first encountered intersecting primitive id (if any) without constructing the corresponding intersection object, and stops after the first encountered intersection. Note that the traversal order of the tree is such that first herein does not refer to any particular ordering of the intersections with respect to the query.

Constructions:

- Function *all_intersections* detects and constructs all intersection objects with the input primitives.
- Function *any_intersection* detects and constructs the first encountered intersection and constructs the corresponding object. This function is fast as it stops after the first encountered intersection.

Distance. An AABB tree computes the closest point from a given point query to the input primitives through the function *closest_point(query)*. In addition, it can compute the id of the closest primitive from a given point query through the function *closest_point_and_primitive(query)*, i.e., the id of the primitive which realizes the minimum distance from the point query. The AABB tree uses a secondary search structure to speed up the distance queries. The construction of this secondary structure should be requested by the user by a call to *accelerate_distance_queries* before the first the distance computation. This data structure is not generated by default because it is used only for distance computations.

64.3 Examples

64.3.1 Tree of Triangles, for Intersection and Distance Queries

In the following example a set of 3D triangles is stored in a list. The AABB primitive wraps a triangle as *datum* and an iterator in the list as *id*. We compute the number of input triangles intersected by a ray query, as well as the closest point and the squared distance from a point query.

```
#include <iostream>
#include <list>

#include <CGAL/Simple_cartesian.h>
#include <CGAL/AABB_tree.h>
#include <CGAL/AABB_traits.h>
#include <CGAL/AABB_triangle_primitive.h>

typedef CGAL::Simple_cartesian<double> K;

typedef K::FT FT;
```

```
typedef K::Ray_3 Ray;
typedef K::Line_3 Line;
typedef K::Point_3 Point;
typedef K::Triangle_3 Triangle;

typedef std::list<Triangle>::iterator Iterator;
typedef CGAL::AABB_triangle_primitive<K,Iterator> Primitive;
typedef CGAL::AABB_traits<K, Primitive> AABB_triangle_traits;
typedef CGAL::AABB_tree<AABB_triangle_traits> Tree;

int main()
{
    Point a(1.0, 0.0, 0.0);
    Point b(0.0, 1.0, 0.0);
    Point c(0.0, 0.0, 1.0);
    Point d(0.0, 0.0, 0.0);

    std::list<Triangle> triangles;
    triangles.push_back(Triangle(a,b,c));
    triangles.push_back(Triangle(a,b,d));
    triangles.push_back(Triangle(a,d,c));

    // constructs AABB tree
    Tree tree(triangles.begin(),triangles.end());

    // counts #intersections
    Ray ray_query(a,b);
    std::cout << tree.number_of_intersected_primitives(ray_query)
        << " intersections(s) with ray query" << std::endl;

    // compute closest point and squared distance
    Point point_query(2.0, 2.0, 2.0);
    Point closest_point = tree.closest_point(point_query);
    std::cerr << "closest point is: " << closest_point << std::endl;
    FT sqd = tree.squared_distance(point_query);
    std::cout << "squared distance: " << sqd << std::endl;

    return EXIT_SUCCESS;
}
```

File: examples/AABB_tree/AABB_triangle_3_example.cpp

64.3.2 Tree of Polyhedron Triangle Facets for Intersection Queries

In the following example the AABB primitive wraps a facet handle of a triangle polyhedral surface as *id* and the corresponding 3D triangle as geometric object. From a segment query we test the intersections, then compute the number of intersections, compute the first encountered intersection (generally a point), compute all intersections (where each intersection is a pair of one CGAL object and one primitive id - here a face handle) and compute all intersected primitives. The latter involves only tests and no predicates and is hence faster than computing all intersections. We also compute the first encountered intersection with a plane query, which is generally a segment.

```
#include <iostream>

#include <CGAL/Simple_cartesian.h>
#include <CGAL/AABB_tree.h>
#include <CGAL/AABB_traits.h>
#include <CGAL/Polyhedron_3.h>
#include <CGAL/AABB_polyhedron_triangle_primitive.h>

typedef CGAL::Simple_cartesian<double> K;
typedef K::Point_3 Point;
typedef K::Plane_3 Plane;
typedef K::Vector_3 Vector;
typedef K::Segment_3 Segment;
typedef CGAL::Polyhedron_3<K> Polyhedron;
typedef CGAL::AABB_polyhedron_triangle_primitive<K,Polyhedron> Primitive;
typedef CGAL::AABB_traits<K, Primitive> Traits;
typedef CGAL::AABB_tree<Traits> Tree;
typedef Tree::Object_and_primitive_id Object_and_primitive_id;
typedef Tree::Primitive_id Primitive_id;

int main()
{
    Point p(1.0, 0.0, 0.0);
    Point q(0.0, 1.0, 0.0);
    Point r(0.0, 0.0, 1.0);
    Point s(0.0, 0.0, 0.0);
    Polyhedron polyhedron;
    polyhedron.make_tetrahedron(p, q, r, s);

    // constructs AABB tree
    Tree tree(polyhedron.facets_begin(),polyhedron.facets_end());

    // constructs segment query
    Point a(-0.2, 0.2, -0.2);
    Point b(1.3, 0.2, 1.3);
    Segment segment_query(a,b);

    // tests intersections with segment query
    if(tree.do_intersect(segment_query))
        std::cout << "intersection(s)" << std::endl;
    else
        std::cout << "no intersection" << std::endl;

    // computes #intersections with segment query
    std::cout << tree.number_of_intersected_primitives(segment_query)
        << " intersection(s)" << std::endl;

    // computes first encountered intersection with segment query
    // (generally a point)
    boost::optional<Object_and_primitive_id> intersection =
        tree.any_intersection(segment_query);
    if(intersection)
    {
```

```
        // gets intersection object
        Object_and_primitive_id op = *intersection;
        CGAL::Object object = op.first;
        Point point;
        if(CGAL::assign(point,object))
            std::cout << "intersection object is a point" << std::endl;
    }

    // computes all intersections with segment query (as pairs object -
   primitive_id)
    std::list<Object_and_primitive_id> intersections;
    tree.all_intersections(segment_query,
   std::back_inserter(intersections));

    // computes all intersected primitives with segment query as
   primitive ids
    std::list<Primitive_id> primitives;
    tree.all_intersected_primitives(segment_query,
   std::back_inserter(primitives));

    // constructs plane query
    Vector vec(0.0,0.0,1.0);
    Plane plane_query(a,vec);

    // computes first encountered intersection with plane query
    // (generally a segment)
    intersection = tree.any_intersection(plane_query);
    if(intersection)
    {
        // gets intersection object
        Object_and_primitive_id op = *intersection;
        CGAL::Object object = op.first;
        Segment segment;
        if(CGAL::assign(segment,object))
            std::cout << "intersection object is a segment" << std::endl;
    }

    return EXIT_SUCCESS;
}
```

File: examples/AABB_tree/AABB_polyhedron_facet_intersection_example.cpp

64.3.3 Tree of Polyhedron Triangle Facets for Distance Queries

In the following example the AABB primitive wraps a facet handle of a triangle polyhedral surface as *id* and the corresponding 3D triangle as geometric object. From a point query we compute the squared distance, the closest point as well as the closest point and primitive id. The latter returns a pair composed of a point and a face handle.

```
// Author(s) : Pierre Alliez
```

```
#include <iostream>

#include <CGAL/Simple_cartesian.h>
#include <CGAL/AABB_tree.h>
#include <CGAL/AABB_traits.h>
#include <CGAL/Polyhedron_3.h>
#include <CGAL/AABB_polyhedron_triangle_primitive.h>

typedef CGAL::Simple_cartesian<double> K;
typedef K::FT FT;
typedef K::Point_3 Point;
typedef K::Segment_3 Segment;
typedef CGAL::Polyhedron_3<K> Polyhedron;
typedef CGAL::AABB_polyhedron_triangle_primitive<K,Polyhedron> Primitive;
typedef CGAL::AABB_traits<K, Primitive> Traits;
typedef CGAL::AABB_tree<Traits> Tree;
typedef Tree::Object_and_primitive_id Object_and_primitive_id;
typedef Tree::Point_and_primitive_id Point_and_primitive_id;

int main()
{
    Point p(1.0, 0.0, 0.0);
    Point q(0.0, 1.0, 0.0);
    Point r(0.0, 0.0, 1.0);
    Point s(0.0, 0.0, 0.0);
    Polyhedron polyhedron;
    polyhedron.make_tetrahedron(p, q, r, s);

    // constructs AABB tree and computes internal KD-tree
    // data structure to accelerate distance queries
    Tree tree(polyhedron.facets_begin(),polyhedron.facets_end());
    tree.accelerate_distance_queries();

    // query point
    Point query(0.0, 0.0, 3.0);

    // computes squared distance from query
    FT sqd = tree.squared_distance(query);
    std::cout << "squared distance: " << sqd << std::endl;

    // computes closest point
    Point closest = tree.closest_point(query);
    std::cout << "closest point: " << closest << std::endl;

    // computes closest point and primitive id
    Point_and_primitive_id pp = tree.closest_point_and_primitive(query);
    Point closest_point = pp.first;
    Polyhedron::Face_handle f = pp.second; // closest primitive id
    std::cout << "closest point: " << closest_point << std::endl;
    std::cout << "closest triangle: ( "
              << f->halfedge()->vertex()->point() << " , "
              << f->halfedge()->next()->vertex()->point() << " , "
              << f->halfedge()->next()->next()->vertex()->point()
              << " )" << std::endl;
```

```
    return EXIT_SUCCESS;
}
```

File: examples/AABB_tree/AABB_polyhedron_facet_distance_example.cpp

64.3.4 Tree of Segments for Intersection and Distance Queries

In the following example the segments are stored into a list, and the AABB primitive wraps a segment as *datum* and an iterator in the list as *id*. We compute the number of intersections with plane and triangles queries, and the closest point from a point query.

```
#include <iostream>
#include <list>

#include <CGAL/Simple_cartesian.h>
#include <CGAL/AABB_tree.h>
#include <CGAL/AABB_traits.h>
#include <CGAL/AABB_segment_primitive.h>

typedef CGAL::Simple_cartesian<double> K;

typedef K::FT FT;
typedef K::Point_3 Point;
typedef K::Plane_3 Plane;
typedef K::Segment_3 Segment;
typedef K::Triangle_3 Triangle;

typedef std::list&lt;Segment&gt;::iterator Iterator;
typedef CGAL::AABB_segment_primitive<K,Iterator> Primitive;
typedef CGAL::AABB_traits<K, Primitive> Traits;
typedef CGAL::AABB_tree<Traits> Tree;

int main()
{
    Point a(1.0, 0.0, 0.0);
    Point b(0.0, 1.0, 0.0);
    Point c(0.0, 0.0, 1.0);
    Point d(0.0, 0.0, 0.0);

    std::list&lt;Segment&gt; segments;
    segments.push_back(Segment(a,b));
    segments.push_back(Segment(a,c));
    segments.push_back(Segment(c,d));

    // constructs the AABB tree and the internal search tree for
    // efficient distance computations.
    Tree tree(segments.begin(),segments.end());
    tree.accelerate_distance_queries();

    // counts #intersections with a plane query
    Plane plane_query(a,b,d);
```

```
    std::cout << tree.number_of_intersected_primitives(plane_query)
        << " intersections(s) with plane" << std::endl;

    // counts #intersections with a triangle query
    Triangle triangle_query(a,b,c);
    std::cout << tree.number_of_intersected_primitives(triangle_query)
        << " intersections(s) with triangle" << std::endl;

    // computes the closest point from a point query
    Point point_query(2.0, 2.0, 2.0);
    Point closest = tree.closest_point(point_query);

    std::cerr << "closest point is: " << closest << std::endl;
    return EXIT_SUCCESS;
}
```

File: examples/AABB_tree/AABB_segment_3_example.cpp

64.3.5 Tree of Polyhedron Edge Segments for Intersection and Distance Queries

In the following example the AABB primitive wraps a halfedge handle as *id* and generates a 3D segment on the fly, each time its method *datum* is called. We compute the number of intersections with a triangle query and the closest point from a point query.

```
#include <iostream>

#include <CGAL/Simple_cartesian.h>
#include <CGAL/AABB_tree.h>
#include <CGAL/AABB_traits.h>
#include <CGAL/Polyhedron_3.h>
#include <CGAL/AABB_polyhedron_segment_primitive.h>

typedef CGAL::Simple_cartesian<double> K;
typedef K::FT FT;
typedef K::Point_3 Point;
typedef K::Triangle_3 Triangle;
typedef CGAL::Polyhedron_3<K> Polyhedron;
typedef CGAL::AABB_polyhedron_segment_primitive<K,Polyhedron> Primitive;
typedef CGAL::AABB_traits<K, Primitive> Traits;
typedef CGAL::AABB_tree<Traits> Tree;

int main()
{
    Point p(1.0, 0.0, 0.0);
    Point q(0.0, 1.0, 0.0);
    Point r(0.0, 0.0, 1.0);
    Point s(0.0, 0.0, 0.0);
    Polyhedron polyhedron;
    polyhedron.make_tetrahedron(p, q, r, s);

    // constructs the AABB tree and the internal search tree for
```

```
    // efficient distance queries.
    Tree tree(polyhedron.edges_begin(),polyhedron.edges_end());
    tree.accelerate_distance_queries();

    // counts #intersections with a triangle query
    Triangle triangle_query(p,q,r);
    std::cout << tree.number_of_intersected_primitives(triangle_query)
        << " intersections(s) with triangle" << std::endl;

    // computes the closest point from a query point
    Point point_query(2.0, 2.0, 2.0);
    Point closest = tree.closest_point(point_query);

    std::cerr << "closest point is: " << closest << std::endl;
    return EXIT_SUCCESS;
}
```

File: examples/AABB_tree/AABB_polyhedron_edge_example.cpp

64.3.6 Incremental Insertion of Primitives

The AABB tree is a static data structure, but it allows to insert primitives, and will internally rebuild triggered by the first query, or because the user calls the *build* method. The following example illustrates this for two polyhedral surfaces.

```
#include <iostream>

#include <CGAL/Simple_cartesian.h>
#include <CGAL/AABB_tree.h>
#include <CGAL/AABB_traits.h>
#include <CGAL/Polyhedron_3.h>
#include <CGAL/AABB_polyhedron_triangle_primitive.h>

typedef CGAL::Simple_cartesian<double> K;
typedef K::FT FT;
typedef K::Point_3 Point;
typedef K::Segment_3 Segment;
typedef CGAL::Polyhedron_3<K> Polyhedron;
typedef CGAL::AABB_polyhedron_triangle_primitive<K,Polyhedron> Primitive;
typedef CGAL::AABB_traits<K, Primitive> Traits;
typedef CGAL::AABB_tree<Traits> Tree;
typedef Tree::Object_and_primitive_id Object_and_primitive_id;
typedef Tree::Point_and_primitive_id Point_and_primitive_id;

int main()
{
    Point p(1.0, 0.0, 0.0);
    Point q(0.0, 1.0, 0.0);
    Point r(0.0, 0.0, 1.0);
    Point s(0.0, 0.0, 0.0);
    Polyhedron polyhedron1;
    polyhedron1.make_tetrahedron(p, q, r, s);
```

```
    Point p2(11.0, 0.0, 0.0);
    Point q2(10.0, 1.0, 0.0);
    Point r2(10.0, 0.0, 1.0);
    Point s2(10.0, 0.0, 0.0);
    Polyhedron polyhedron2;
    polyhedron2.make_tetrahedron(p2, q2, r2, s2);
    // constructs AABB tree and computes internal KD-tree
    // data structure to accelerate distance queries
    Tree tree(polyhedron1.facets_begin(),polyhedron1.facets_end());

    tree.accelerate_distance_queries();

    tree.insert(polyhedron2.facets_begin(),polyhedron2.facets_end());

    // query point
    Point query(0.0, 0.0, 3.0);

    // computes squared distance from query
    FT sqd = tree.squared_distance(query);
    std::cout << "squared distance: " << sqd << std::endl;

    return EXIT_SUCCESS;
}
```

File: examples/AABB_tree/AABB_insertion_example.cpp

64.3.7 Trees of Custom Primitives

The AABB tree example folder contains three examples of trees constructed with customize primitives. In *AABB_custom_example.cpp* the primitive contains triangles which are defined by three pointers to custom points. In *AABB_custom_triangle_soup_example.cpp* all input triangles are stored into a single array so as to form a triangle soup. The primitive internally uses a *boost::iterator_adaptor* so as to provide the three functions (*id()*, *datum()*, *reference_point()*) required by the primitive concept. In *AABB_custom_indexed_triangle_set_example.cpp* the input is an indexed triangle set stored through two arrays: one array of points and one array of indices which refer to the point array. Here also the primitive internally uses a *boost::iterator_adaptor*.

64.4 Performances

We provide some performance numbers for the case where the AABB tree contains a set of polyhedron triangle facets. We measure the tree construction time, the memory occupancy and the number of queries per second for a variety of intersection and distance queries. The machine used is a PC running Windows XP64 with an Intel CPU Core2 Extreme clocked at 3.06 GHz with 4GB of RAM. By default the kernel used is *Simple_cartesian< double>* (the fastest in our experiments). The program has been compiled with Visual C++ 2005 compiler with the O2 option which maximizes speed.

64.4.1 Construction

The surface triangle mesh chosen for benchmarking the tree construction is the knot model (14,400 triangles) depicted by Figure 64.4.3. We measure the tree construction time (both AABB tree alone and AABB tree with internal KD-tree) for this model as well as for three denser versions subdivided through the Loop subdivision scheme which multiplies the number of triangles by four.

Number of triangles	Construction (in ms)	Construction with internal KD-tree (in ms)
14,400	156	157
57,600	328	328
230,400	1,141	1,437
921,600	4,813	5,953

64.4.2 Memory

When using the polyhedron triangle facet primitive (defined in *AABB_polyhedron_triangle_primitive.h*) the AABB tree occupies approximately 61 bytes per primitive (without constructing the internal KD-tree). It increases to approximately 150 bytes per primitive when constructing the internal KD-tree with one reference point per primitive (the default mode when calling the function *tree.accelerate_distance_queries()*). Note that the polyhedron facet primitive primitive stores only one facet handle as primitive id and computes on the fly a 3D triangle from the facet handle stored internally. When explicitly storing a 3D triangle in the primitive the tree occupies approximately 140 bytes per primitive instead of 60 (without constructing the internal KD-tree).

The following table provides orders of memory occupancy in MBytes for an increasing number of triangles. As the internal KD-tree used to accelerate the distance queries dominates the memory occupancy, we recommend to specify for large models a lower number of reference point (evenly distributed) to construct the internal KD-tree through the function *tree.accelerate_distance_queries(begin,end)* which takes an iterator range as input.

Number of triangles	AABB tree (in MBytes)	AABB tree with internal KD-tree (in MBytes)
18,400	1.10	2.76
102,400	6.33	14.73
1,022,400	59.56	151.31
1,822,400	108.34	291.84

64.4.3 Intersections

The following table measures the number of intersection queries per second on the 14,400 triangle version of the knot mesh model for ray, line, segment and plane queries. Each ray query is generated by choosing a random source point within the mesh bounding box and a random vector. A line or segment query is generated by choosing two random points inside the bounding box. A plane query is generated by picking a random point inside the bounding box and a random normal vector. Note that a plane query generally intersects many triangles of the input surface mesh. This explains the low performance numbers for the intersection functions which enumerate all intersections.

Function	Segment	Ray	Line	Plane
do_intersect()	187,868	185,649	206,096	377,969
any_intersected_primitive()	190,684	190,027	208,941	360,337
any_intersection()	147,468	143,230	148,235	229,336
number_of_intersected_primitives()	64,389	52,943	54,559	7,906
all_intersected_primitives()	65,553	54,838	53,183	5,693
all_intersections()	46,507	38,471	36,374	2,644

Curve 64.4.3 plots the number of queries per second (here the *all_intersections* function with random segment queries) against the number of input triangles for the knot triangle surface mesh.

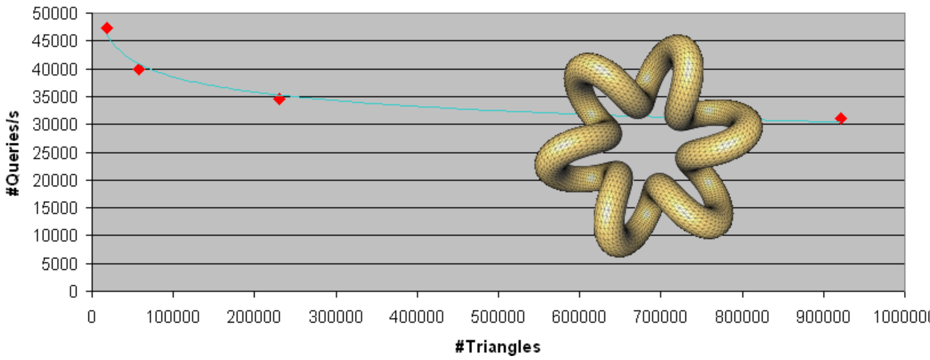


Figure 64.2: Number of queries per second against number of triangles for the knot model with 14K (shown), 57K, 230K and 921K triangles. We call the *all_intersections* function with segment queries randomly chosen within the bounding box.

The following table measures the number of *all_intersections()* queries per second against several kernels. We use the 14,400 triangle version of the knot mesh model for random segment queries. Note how the *Simple_cartesian* kernel is substantially faster than the *Cartesian* kernel.

Kernel	Queries/s (all_intersections() with segment queries)
Simple_cartesian<double>	46,507
Simple_cartesian<float>	43,187
Cartesian<double>	5,335
Cartesian<float>	5,522
Exact_predicates_inexact_constructions	18,411

64.4.4 Distances

The surface triangle mesh chosen for benchmarking distances is again the knot model in four increasing resolutions obtained through Loop subdivision. In the following table we first measure the tree construction time (which includes the construction of the internal KD-tree data structure used to accelerate the distance queries by up to one order of magnitude in our experiments). We then measure the number of queries per second for the three types distance queries (*closest_point*, *squared_distance* and *closest_point_and_primitive*) from point queries randomly chosen inside the bounding box.

Nb triangles	Construction (ms)	Closest_point()	squared_distance()	closest_point_and_primitive()
14,400	157	45,132	45,626	45,770
57,600	328	21,589	21,312	21,137
230,400	1,437	11,063	10,962	11,086
921,600	5,953	5,636	5,722	5,703

64.4.5 Summary

The experiments described above are neither exhaustive nor conclusive as we have chosen one specific case where the input primitives are the facets of a triangle surface polyhedron. Nevertheless we now provide some general observations and advices about how to put the AABB tree to use with satisfactory performances. While the tree construction times and memory occupancy do not fluctuate much in our experiments depending on the input surface triangle mesh, the performance expressed in number of queries varies greatly depending on a complex combination of criteria: type of kernel, number of input primitives, distribution of primitives in space, type of function queried, type of query and location of query in space.

- Kernel: The type of CGAL kernel turns out to dominate the final execution times, the maximum performances being obtained with the simple Cartesian kernel parameterized with the double precision number type. In applications where the intersection and distance execution times are crucial it is possible to use this kernel for the AABB tree in combination with a more robust kernel for the main data structure.
- Primitives: Although the number of input primitives plays an obvious role in the final performance, their distribution in space is at least equally important in order to obtain a well-balanced AABB tree. Ideally the primitives must be evenly distributed in space and the long primitives spanning the bounding box of the tree root node must be avoided as much as possible. It is often beneficial to split these long primitives into smaller ones before constructing the tree, e.g., through recursive longest edge bisection for triangle surface meshes.
- Function: The type of function queried plays another important role. Obviously the "exhaustive" functions, which list all intersections, are slower than the ones stopping after the first intersection. Within each of these functions the ones which call only intersection tests (do_intersect(), number_of_intersected_primitives(), any_intersected_primitive(), all_intersected_primitives()) are faster than the ones which explicitly construct the intersections (any_intersection() and all_intersections()).
- Query: The type of query (e.g., line, ray, segment or plane used above) plays another role, strongly correlated with the type of function (exhaustive or not, and whether or not it constructs the intersections). When all intersection constructions are needed, the final execution times highly depend on the complexity of the general intersection object. For example a plane query generally intersects a surface triangle mesh into many segments while a segment query generally intersects a surface triangle mesh into few points. Finally, the location of the query in space also plays an obvious role in the performances, especially for the distance queries. Assuming the internal KD-tree constructed through the function *tree.accelerate_distance_queries()*, it is preferable to specify a query point already close to the surface triangle mesh so that the query traverses only few AABBs of the tree. For a large number of primitive data (greater than 2M faces in our experiments) however we noticed that it is not necessary (and sometimes even slower) to use all reference points when constructing the KD-tree. In these cases we recommend to specify trough the function *tree.accelerate_distance_queries(begin,end)* fewer reference points (typically not more than 100K) evenly distributed over the input primitives.

64.5 Implementation Details

The AABB tree construction is initialized by computing the AABB of the whole set of input primitives. All primitives are then sorted along the longest coordinate axis of this box, and the primitives are separated into two equal size sets. This procedure is applied recursively until an AABB contains a single primitive. The tree is leafless as presented in OPCODE [Ter05]. An intersection query traverses the tree by computing intersection tests only with respect to the AABBs during traversal, and with respect to the input primitives at the end of traversal (in the leafs of the tree).

The reference id is not used internally but simply used by the AABB tree to refer to the primitive in the results provided to the user. It follows that, while in most cases each reference id corresponds to a unique primitive, this is not a requirement of the component. This way a user may use these reference ids as labels, each of them being shared by several geometric object.

A distance query between a query point q and the input primitives is turned into a *ball* query centered at q. The ball traverses the tree while recursively querying intersections with the AABBs, and computes the closest point p from the query point to the input primitives at the leafs of the tree. The ball radius is then shrunk to the distance between p and q for all remaining recursive traversals of the tree. Efficiency is achieved through setting the initial ball radius to a small value still guaranteed to intersect the input primitives. This is achieved by constructing through the function *accelerate_distance_queries* an internal secondary data structure which provides a good hint to the algorithm at the beginning of the traversal.

64.6 Design and Implementation History

Camille Wormser and Pierre Alliez started working on a data structure for efficient collision detection in 2007. The generic design for implementing both intersection and distance queries, and for generic queries and primitives was developed by Camille Wormser. In 2009, Pierre Alliez, Stéphane Tayeb and Camille Wormser made the implementation CGAL-compliant, with the help of Laurent Rineau for optimizing the tree construction. The authors wish to thank Andreas Fabri, Jane Tournois, Mariette Yvinec and Sylvain Lefèbvre for helpful comments and discussions.

3D Fast Intersection and Distance Computation (AABB Tree) Reference Manual

Pierre Alliez, Stéphane Tayeb, Camille Wormser

64.7 Classified Reference Pages

Concepts

Classes

64.8 Alphabetical List of Reference Pages

Class

CGAL::AABB_tree<AT>

Definition

Class *AABB_tree<AT>* is a static data structure for efficient intersection and distance computations in 3D. It builds a hierarchy of axis-aligned bounding boxes (an AABB tree) from a set of 3D geometric objects, and can receive intersection and distance queries, provided that the corresponding predicates are implemented in the traits class AT. The template parameter AT stands for a traits class which must be a model of the concept *AABBTraits*.

#include <CGAL/AABB_tree.h>

Types

AABB_tree<AT>:: size_type;		Unsigned integral size type.
typedef AT::FT	*FT;*	Number type returned by the distance queries.
typedef AT::Point_3	*Point;*	Type of 3D point.
typedef AT::Primitive	*Primitive;*	Type of input primitive.
typedef AT::Bounding_box	*Bounding_box;*	Type of bounding box.
typedef std::pair<Point, Primitive::Id>	*Point_and_primitive_id;*	
typedef std::pair<CGAL::Object, Primitive::Id>	*Object_and_primitive_id;*	

Creation

AABB_tree<AT> tree; Constructs an empty tree.

template < class InputIterator>
AABB_tree<AT> tree(InputIterator begin, InputIterator beyond);

Builds the AABB tree data structure. Type *InputIterator* can be any const iterator such that *Primitive* has a constructor taking a *InputIterator* as argument. The tree stays empty if the memory allocation is not successful.

Operations

template < class InputIterator>
void *tree.rebuild(InputIterator begin, InputIterator beyond)*

Clears the current tree and rebuilds it from scratch. See constructor above for the parameters.

void *tree.clear()* Clears the AABB tree.

template <class InputIterator>
void *tree.insert(InputIterator begin, InputIterator beyond)*

Add a sequence of primitives to the set of primitive of the AABB tree. Type *InputIterator* can be any const iterator such that *Primitive* has a constructor taking an InputIterator as argument.

void *tree.insert(const Primitive p)*

Add a primitive to the set of primitives of the AABB tree.

advanced

void *tree.build()* After one or more calls to *insert*, the internal data structure of *AABB_tree<AT>* must be reconstructed. This procedure has a complexity of *O(n log(n))*, where n is the number of primitives of the tree. This procedure is called implicitly at the first call to a query member function. You can call *build()* explicitly to ensure that the next call to query functions will not trigger the reconstruction of the data structure.

advanced

Bounding_box *tree.bbox() const* Returns the axis-aligned bounding box of the whole tree.
size_type *tree.size() const* Returns the number of primitives in the tree.
bool *tree.empty() const* Returns *true*, iff tree contains no primitive.

Intersection Tests

template <class Query>
bool *tree.do_intersect(Query query) const*

Returns *true*, iff the query intersects at least one of the input primitives. Type *Query* must be a type for which *do_intersect* predicates are defined in the *AT* class.

template <class Query>
size_type *tree.number_of_intersected_primitives(Query query) const*

Returns the number of primitives intersected by the query. Type *Query* must be a type for which *do_intersect* predicates are defined in the *AT* class.

template <class Query, class OutputIterator>
OutputIterator *tree.all_intersected_primitives(Query query, OutputIterator out) const*

Outputs to the iterator the list of all intersected primitives ids. This function does not compute the intersection points and is hence faster than the function *all_intersections* function below. Type *Query* must be a type for which *do_intersect* predicates are defined in the *AT* class.

template <class Query>

boost::optional<Primitive::Id>

tree.any_intersected_primitive(Query query) const

Returns the first encountered intersected primitive id, iff the query intersects at least one of the input primitives. No particular order is guaranteed over the tree traversal, such that, e.g, the primitive returned is not necessarily the closest from the source point of a ray query. Type *Query* must be a type for which *do_intersect* predicates are defined in the *AT* class.

Intersections

template <class Query, class OutputIterator>
OutputIterator *tree.all_intersections(Query query, OutputIterator out) const*

Outputs to the iterator the list of all intersections between the query and input data, as objects of type *Object_and_primitive_id*. Type *Query* must be a type for which *do_intersect* predicates and intersections are defined in the *AT* class.

template <class Query>
boost::optional<Object_and_primitive_id>

tree.any_intersection(Query query) const

Returns the first encountered intersection, iff the query intersects at least one of the input primitives. No particular order is guaranteed over the tree traversal, such that, e.g, the primitive returned is not necessarily the closest from the source point of a ray query. Type *Query* must be a type for which *do_intersect* predicates and intersections are defined in the *AT* class.

Distance Queries

FT *tree.squared_distance(Point query) const*

Returns the minimum squared distance between the query point and all input primitives. Method *accelerate_distance_queries* should be called before the first distance query, so that an internal secondary search structure is build, for improving performance.

Point *tree.closest_point(Point query) const*

Returns the point in the union of all input primitives which is closest to the query. In case there are several closest points, one arbitrarily chosen closest point is returned. Method *accelerate_distance_queries* should be called before the first distance query, so that an internal secondary search structure is build, for improving performance.

Point_and_primitive_id

tree.closest_point_and_primitive(Point query) const

Returns a *Point_and_primitive_id* which realizes the smallest distance between the query point and all input primitives. Method *accelerate_distance_queries* should be called before the first distance query, so that an internal secondary search structure is build, for improving performance.

Accelerating the Distance Queries

bool *tree.accelerate_distance_queries() const*

Constructs an internal data structure for accelerating distance queries. This method should be called once, before the first distance query. Returns *true*, iff the memory allocation is successful.

advanced

In the following paragraphs, we discuss details of the implementation of the distance queries. We explain the internal use of hints, how the user can pass his own hints to the tree, and how the user can influence the construction of the secondary data structure used for accelerating distance queries.

Internally, the distance queries algorithms are initialized with some hint, which has the same type as the return type of the query, and this value is refined along a traversal of the tree, until it is optimal, that is to say until it realizes the shortest distance to the primitives. In particular, the exact specification of these internal algorithms is that they minimize the distance to the object composed of the union of the primitives and the hint.

It follows that

- in order to return the exact distance to the set of primitives, the algorithms need the hint to be exactly on the primitives;
- if this is not the case, and if the hint happens to be closer to the query point than any of the primitives, then the hint is returned.

This second observation is reasonable, in the sense that providing a hint to the algorithm means claiming that this hint belongs to the union of the primitives. These considerations about the hints being exactly on the primitives or not are important: in the case where the set of primitives is a triangle soup, and if some of the primitives are large, one may want to provide a much better hint than a vertex of the triangle soup could be. It could be, for example, the barycenter of one of the triangles. But, except with the use of an exact constructions kernel, one cannot easily construct points other than the vertices, that lie exactly on a triangle soup. Hence, providing a good hint sometimes means not being able to provide it exactly on the primitives. In rare occasions, this hint can be returned as the closest point.

In order to accelerate distance queries significantly, the AABB tree builds an internal KD-tree containing a set of potential hints, when the method *accelerate_distance_queries* is called. This KD-tree provides very good hints that allow the algorithms to run much faster than with a default hint (such as the *reference_point* of the first primitive). The set of potential hints is a sampling of the union of the primitives, which is obtained, by default, by calling the method *reference_point* of each of the primitives. However, such a sampling with one point per primitive may not be the most relevant one: if some primitives are very large, it helps inserting more than one sample on them. Conversely, a sparser sampling with less than one point per input primitive is relevant in some cases.

For this reason, the user can provide his own set of sample points:

template <class InputIterator>
bool *tree.accelerate_distance_queries(InputIterator begin, InputIterator beyond) const*

Constructs an internal KD-tree containing the specified point set, to be used as the set of potential hints for accelerating the distance queries. *InputIterator* is an iterator with value type *Point_and_primitive_id*.

Note that, in some cases, the user is not able to provide ids of the primitives on which the points lie. In these cases, providing a default value for the ids of the hints is possible. Still, the user should be aware that if she uses the *closest_point_and_primitive* method, there is a (tiny) chance that a hint is returned, along with this default value as corresponding primitive id. Hence, the validity of the returned primitive id should be checked in these cases.

As an alternative to using the KD-tree, the user can also provide the hints directly, by using the following methods:

FT *tree.squared_distance(Point query, Point hint) const*

Returns the minimum squared distance between the query point and all input primitives. The internal KD-tree is not used.

Point *tree.closest_point(Point query, Point hint) const*

Returns the point in the union of all input primitives which is closest to the query. In case there are several closest points, one arbitrarily chosen closest point is returned. The internal KD-tree is not used.

Point_and_primitive_id

tree.closest_point_and_primitive(Point query, Point_and_primitive_id hint) const

Returns a *Point_and_primitive_id* which realizes the smallest distance between the query point and all input primitives. The internal KD-tree is not used.

advanced

See Also

AABBTraits,
AABBPrimitive.

AABBGeomTraits

Concept

Definition

The concept AABBGeomTraits defines the requirements for the first template parameter of the class *AABB_traits<GeomTraits, Primitive>*. It provides predicates and constructors to detect and compute intersections between query objects and the primitives stored in the AABB tree. In addition, it contains predicates and constructors to compute distances between a point query and the primitives stored in the AABB tree.

Types

AABBGeomTraits:: Bbox_3	Bounding box type.
AABBGeomTraits:: Sphere_3	Sphere type, that should be consistent with the distance function chosen for the distance queries, namely the *Squared_distance_3* functor.
AABBGeomTraits:: Point_3	Point type.
AABBGeomTraits:: Do_intersect_3	A functor object to detect intersections between two geometric objects. Provides the operators: *bool operator()(const Type_1& type_1, const Type_2& type_2);* where *Type_1* and *Type_2* are relevant types among *Ray_3*, *Segment_3*, *Line_3*, *Triangle_3*, *Plane_3* and *Bbox_3*. Relevant herein means that a line primitive (ray, segment, line) is tested against a planar or solid primitive (plane, triangle, box), and a solid primitive is tested against another solid primitive (box against box). The operator returns *true* iff *type_1* and *type_2* have a non empty intersection.
AABBGeomTraits:: Intersect_3	A functor object to construct the intersection between two geometric objects. Provides the operators: *CGAL::Object operator()(const Type_1& type_1, const Type_2& type_2);* where *Type_1* and *Type_2* are any relevant types among *Ray_3*, *Segment_3*, *Line_3*, *Triangle_3*, *Plane_3* and *Bbox_3*. Relevant herein means that a line primitive (ray, segment, line) is tested against a planar or solid primitive (plane, triangle, box). The operator computes a *CGAL::Object* which is the intersection between the objects *type_1* and *type_2*.
AABBGeomTraits:: Construct_sphere_3	A functor object to construct the sphere centered at one point and passing through another one. Provides the operator: *Sphere_3 operator()(const Point_3& p, const Point_3 & q);* which returns the sphere centered at *p* and passing through *q*.
AABBGeomTraits:: Compute_closest_point_3	A functor object to compute the point on a geometric primitive which is closest from a query. Provides the operator: *Point_3 operator()(const Point_3& p, const Type_2& type_2);* where *Type_2* is any type among *Segment_3* and *Triangle_3*. The operator returns the point on *type_2* which is closest to *p*.

AABBGeomTraits:: Has_on_bounded_side_3 A functor object to detect if a point lies inside a sphere or not. Provides the operator: *bool operator()(const Sphere_3& s, const Point_3& p);* which returns *true* iff the closed volume bounded by *s* contains *p*.

AABBGeomTraits:: Compute_squared_radius_3

A functor object to compute the squared radius of a sphere. Provides the operator: *FT operator()(const Sphere_3& s);* which returns the squared radius of *s*.

Creation

Operations

Do_intersect_3 *geomtraits.do_intersect_3_object()*

Returns the intersection detection functor.

Intersect_3 *geomtraits.intersect_3_object()*

Returns the intersection constructor.

Construct_sphere_3 *geomtraits.construct_sphere_3_object()*

Returns the distance comparison functor.

Compute_closest_point_3

geomtraits.compute_closest_point_3_object()

Returns the closest point constructor.

Has_on_bounded_side_3

geomtraits.has_on_bounded_side_3_object()

Returns the closest point constructor.

Compute_squared_radius_3

geomtraits.compute_squared_radius_3_object()

Returns the squared radius functor.

Has Models

Any instantiation of *CGAL::Kernel* is a model of this traits concept.

See Also

AABB_traits<GeomTraits,AABBPrimitive>.

Concept

AABBPrimitive

Definition

The concept AABBPrimitive describes the requirements for the primitives stored in the AABB tree data structure. The concept encapsulates a type for the input datum (a geometric object) and an identifier (id) type through which those primitives are referred to. The concept AABBPrimitive also refines the concepts DefaultConstructible and Assignable.

Types

AABBPrimitive:: Point	3D point type.
AABBPrimitive:: Datum	Type of input datum.
AABBPrimitive:: Id	Type of identifiers through which the input objects are referred to. It must be a model of the concepts DefaultConstructible and Assignable.

Operations

Datum	*primitive.datum()*	Returns the datum (geometric object) represented by the primitive.
Id	*primitive.id()*	Returns the corresponding identifier. This identifier is only used as a reference for the objects in the output of the *AABB_tree* methods.
Point	*primitive.reference_point()*	Returns a 3D point located on the geometric object represented by the primitive. This function is used to sort the primitives during the AABB tree construction as well as to construct the search KD-tree internal to the AABB tree used to accelerate distance queries.

See Also

AABB_tree<AT>

Example

The *Primitive* type can be, e.g., a wrapper around a *Handle*. Assume for instance that the input objects are the triangle faces of a mesh stored as a *CGAL::Polyhedron*. The *Datum* would be a *Triangle_3* and the *Id* would be a polyhedron *Face_handle*. Method *datum()* can return either a *Triangle_3* constructed on the fly from the face handle or a *Triangle_3* stored internally. This provides a way for the user to trade memory for efficiency.

Has Models

AABB_polyhedron_triangle_primitive<GeomTraits,Polyhedron>,
AABB_polyhedron_segment_primitive<GeomTraits,Polyhedron>.

Class

CGAL::AABB_polyhedron_triangle_primitive<GeomTraits,Polyhedron>

Definition

The class *AABB_polyhedron_triangle_primitive<GeomTraits,Polyhedron>* is a model of the concept *AABBPrimitive*. It wraps a facet handle of a polyhedron, which is used as id, and allows the construction of the datum on the fly. Since only the facet handle is stored in this primitive, the polyhedron from which the AABB tree is built should not be deleted while the AABB tree is in use.

Parameters

The template parameter *GeomTraits* provides a *Point_3* type, used as *Point*, and a *Triangle_3* type, used as *Datum* and constructible from three arguments of type *Point*. The template parameter *Polyhedron* should be a *CGAL::Polyhedron* whose points have type *Point*.

#include <CGAL/AABB_polyhedron_triangle_primitive.h>

Types

typedef GeomTraits::Triangle_3 *Datum;*	Geometric data type.
typedef Polyhedron::Facet_handle *Id;*	Id type.

See Also

AABBPrimitive,
AABB_polyhedron_segment_primitive<GeomTraits,Polyhedron>

CGAL::AABB_polyhedron_segment_primitive<GeomTraits,Polyhedron>

Class

Definition

The class *AABB_polyhedron_segment_primitive<GeomTraits,Polyhedron>* is a model of the concept *AABBPrimitive*. It wraps a halfedge handle of a polyhedron, which is used as id, and allows the construction of the datum on the fly. Since only the halfedge handle is stored in this primitive, the polyhedron from which the AABB tree is built should not be deleted while the AABB tree is in use.

Parameters

The template parameter *GeomTraits* provides a *Point_3* type, used as *Point*, and a *Segment_3* type, used as *Datum* and constructible from two arguments of type *Point*. The template parameter *Polyhedron* should be a *CGAL::Polyhedron* whose points have type *Point*.

#include <CGAL/AABB_polyhedron_segment_primitive.h>

Types

typedef GeomTraits::Segment_3 *Datum;*	Geometric data type.
typedef Polyhedron::Halfedge_handle *Id;*	Id type.

See Also

AABBPrimitive,
AABB_polyhedron_triangle_primitive<GeomTraits,Polyhedron>

AABBTraits

Definition

The concept AABBTraits provides the geometric primitive types and methods for the class *AABB_tree<AT>*.

Types

AABBTraits:: FT — Value type of the *Squared_distance* functor.

AABBTraits:: Point_3 — Type of a 3D point.

AABBTraits:: Primitive — Type of primitive. Must be a model of the concept *AABBPrimitive*.

AABBTraits:: Bounding_box — Bounding box type.

AABBTraits:: enum Axis { X_AXIS, Y_AXIS, Z_AXIS }

typedef std::pair<Point_3, Primitive::Id>

Point_and_primitive_id;

typedef std::pair<Object, Primitive::Id>

Object_and_primitive_id;

During the construction of the *AABB_tree*, the primitives are sorted according to some comparison functions related to the x, y or z coordinate axis:

AABBTraits:: Split_primitives_along_x_axis — A functor object to split a range of primitives into two sub-ranges along the X-axis. Provides the operator: *void operator()(InputIterator first, InputIterator beyond);* Iterator *InputIterator* must be a model of RandomAccessIterator and have *Primitive* as value type. The operator is used for determining the primitives assigned to the two children nodes of a given node, assuming that the goal is to split the X-dimension of the bounding box of the node. The primitives assigned to this node are passed as argument to the operator. It should modify the iterator range in such a way that its first half and its second half correspond to the two children nodes.

AABBTraits:: Split_primitives_along_y_axis — A functor object to split a range of primitives into two sub-ranges along the Y-axis. See *Split_primitives_along_x_axis* for the detailed description.

AABBTraits:: Split_primitives_along_z_axis — A functor object to split a range of primitives into two sub-ranges along the Z-axis. See *Split_primitives_along_x_axis* for the detailed description.

AABBTraits:: Compute_bbox — A functor object to compute the bounding box of a set of primitives. Provides the operator: *Bounding_box operator()(Input_iterator begin, Input_iterator beyond);* Iterator *InputIterator* must have *Primitive* as value type.

AABBTraits:: Splitting_direction — A functor object to specify the direction along which the bounding box should be split: *Axis operator()(const Bounding_box& bbox);* which returns the direction used for splitting *bbox*. It is usually the axis aligned with the longest edge of *bbox*.

The following predicates are required for each type *Query* for which the class *AABB_tree<AT>* may receive an intersection detection or computation query.

AABBTraits:: Do_intersect — A functor object to compute intersection predicates between the query and the nodes of the tree. Provides the operators: *bool operator()(const Query & q, const Bounding_box & box);* which returns *true* iff the query intersects the bounding box, and *bool operator()(const Query & q, const Primitive & primitive);* which returns *true* iff the query intersects the primitive.

AABBTraits:: Intersect — A functor object to compute the intersection of a query and a primitive. Provides the operator: *boost::optional<Object_and_primitive_id> operator()(const Query & q, const Primitive& primitive);* which returns the intersection as a pair composed of an object and a primitive id, iff the query intersects the primitive.

The following predicates are required for each type *Query* for which the class *AABB_tree<AT>* may receive a distance query.

AABBTraits:: Compare_distance — A functor object to compute distance comparisons between the query and the nodes of the tree. Provides the operators: *bool operator()(const Query & query, const Bounding_box& box, const Point & closest);* which returns *true* iff the bounding box is closer to *query* than *closest* is, and *bool operator()(const Query & query, const Primitive & primitive, const Point & closest);* which returns *true* iff *primitive* is closer to the *query* than *closest* is.

AABBTraits:: Closest_point — A functor object to compute closest point from the query on a primitive. Provides the operator: *Point_3 operator()(const Query& query, const Primitive& primitive, const Point_3 & closest);* which returns the closest point to *query*, among *closest* and all points of the primitive.

AABBTraits:: Squared_distance — A functor object to compute the squared distance between the query and a point. Provides the operator: *FT operator()(const Query& query, const Point_3 & p);* which returns the squared distance between *query* and *p*.

Operations

Split_primitives_along_x_axis

traits.split_primitives_along_x_axis_object()

Returns the primitive splitting functor for the X axis.

Split_primitives_along_y_axis

traits.split_primitives_along_y_axis_object()

Returns the primitive splitting functor for the Y axis.

Split_primitives_along_z_axis

traits.split_primitives_along_z_axis_object()

Returns the primitive splitting functor for the Z axis.

Compute_bbox *traits.compute_bbox_object()*

Returns the bounding box constructor.

Do_intersect *traits.do_intersect_object()*

Returns the intersection detection functor.

Intersect *traits.intersect_object()*

Returns the intersection constructor.

Compare_distance *traits.compare_distance_object()*

Returns the distance comparison functor.

Closest_point *traits.closest_point_object()*

Returns the closest point constructor.

Has Models

AABB_traits<GeomTraits,Primitive>.

See Also

AABB_traits<GeomTraits,Primitive>
AABB_tree<AT> AABBPrimitive

CGAL::AABB_traits<GeomTraits,Primitive>

Definition

The class *AABB_traits<GeomTraits,Primitive>* is a model of the concept *AABBTraits*. This traits class handles any type of 3D geometric primitives provided that the proper intersection tests and constructions are implemented. It handles points, rays, lines and segments as query types for intersection detection and computations, and it handles points as query type for distance queries.

Parameters

The template parameter *GeomTraits* provides the geometric types as well as the intersection tests and computations required. This type must be a model of the concept *AABBGeomTraits*. The template parameter *Primitive* provides the type of primitives stored in the AABB tree. This parameter must be a model of the concept *AABBPrimitive*.

#include <CGAL/AABB_traits.h>

Types

typedef GeomTraits::Point_3

Point_3; — Point query type.

typedef GeomTraits::Ray_3

Ray_3; — Ray query type.

typedef GeomTraits::Line_3

Line_3; — Line query type.

typedef GeomTraits::Segment_3

Segment_3; — Segment query type.

AABB_traits<GeomTraits,Primitive>:: size_type

Unsigned integral type.

AABB_traits<GeomTraits,Primitive> traits(GeomTraits gt = GeomTraits());

Default constructor.

See Also

AABBTraits,
AABB_tree<AT>,
AABBPrimitive

Chapter 65

Spatial Sorting

Christophe Delage and Olivier Devillers

Contents

65.1 Introduction

Many geometric algorithms implemented in CGAL are incremental, and thus their speed is dependent on the order of insertion. This package provides sorting algorithms that may considerably improve running times of such algorithms.

The rationale is to sort objects along a space-filling curve so that two objects close geometrically will be close in the insertion order with high probability. That way, parts of a data structure that will be looked at during an insertion will probably have been looked at in a recent insertion, and thus probably will be in cache memory instead of main memory. As another side-effect, these sorting functions usually improve memory locality of the data structures produced by incremental algorithms, sometimes leading to speed ups in other algorithm using these data structures.

Some algorithms have a good complexity under randomized hypotheses which contradicts the idea of sorting the input using any sorting criterion. In such a case, it is possible to introduce just a bit of randomness to be able to combine the good randomized complexity and the good effects of locality [ACR03].

The predicates used by this package are comparisons between coordinates, thus there is no robustness issue involved here, for example to choose the arithmetic of the kernel.

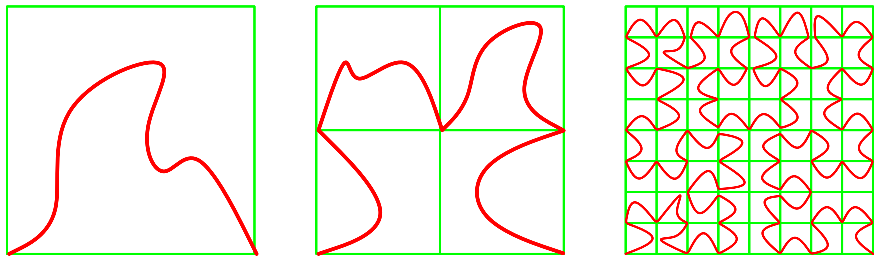


Figure 65.1: Hilbert mapping

65.2 Hilbert Sorting

In 2D, one can construct a space filling curve, that is a mapping f of $[0,1]$ to the unit square $[0,1]^2$, such that $f(0) = (0,0)$ and $f(1) = (1,0)$ in the following way: the unit square is subdivided in four such that

$f([0,\frac{1}{4}]) = [0,\frac{1}{2}]^2$, $f([\frac{1}{4},\frac{1}{2}]) = [0,\frac{1}{2}] \times [\frac{1}{2},1]$, $f([\frac{1}{2},\frac{3}{4}]) = [\frac{1}{2},1]^2$, and $f([\frac{3}{4},1]) = [\frac{1}{2},1] \times [0,\frac{1}{2}]$.
$f(\frac{1}{4}) = (0,\frac{1}{2})$
$f(\frac{1}{2}) = (\frac{1}{2},\frac{1}{2})$, and $f(\frac{3}{4}) = (1,\frac{1}{2})$.

Then each square is subdivided in the same way recursively. Figure 65.1 illustrates this process.

Now given a set of 2D points, they can be sorted in the order they have on such a space filling curve as illustrated in Figure 65.2 :

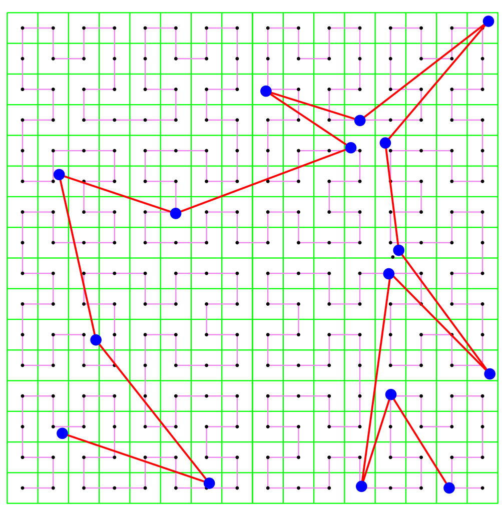


Figure 65.2: Hilbert sort with middle policy

65.2.1 Examples

The code to use Hilbert sort is as simple as the following example:

```
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/point_generators_2.h>
#include <CGAL/hilbert_sort.h>
#include <iostream>
#include <vector>

typedef CGAL::Exact_predicates_inexact_constructions_kernel K;
typedef K::Point_2                                          Point;
typedef CGAL::Creator_uniform_2<double,Point>               Creator;

int main ()
{
  std::size_t size = 16;
  std::vector<Point> v; v.reserve(size);
  CGAL::points_on_square_grid_2(3.0, size,           // generate points
                               std::back_inserter(v), Creator());

  CGAL::hilbert_sort (v.begin(), v.end());           // sort

  for(std::size_t i=0; i<size; ++i)std::cout<<v[i]<<std::endl;//output
  return 0;
}
```

File: `examples/Spatial_sorting/hilbert.cpp`

If instead of subdividing the square in a fixed way at its middle point, as above, we subdivide it by splitting at the median point (in x or y directions alternating), we construct a 2-d tree adapted to the point set. This tree can be visited in a similar manner and we get also a suitable ordering of the points (see Figure 65.3).

CGAL provides Hilbert sorting for points in 2D, 3D and higher dimensions, in the middle and the median policies.

The middle policy is easier to analyze, and is interesting in practice for well distributed set of points in small dimension (if the number of points is really smaller than 2^d). The median policy should be prefered for high dimension or if the point set distribution is not regular (or unknown). Since the median policy cannot be much worse than the middle policy, while the converse can happen, the median policy is the default behavior. Most theoretical results are using the middle policy [ACR03, But71, BG89, PB89].

This other example illustrates the use of the two different policies

```
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/hilbert_sort.h>
#include <iostream>
#include <vector>

typedef CGAL::Exact_predicates_inexact_constructions_kernel K;
typedef K::Point_2                                          Point;

int main ()
```

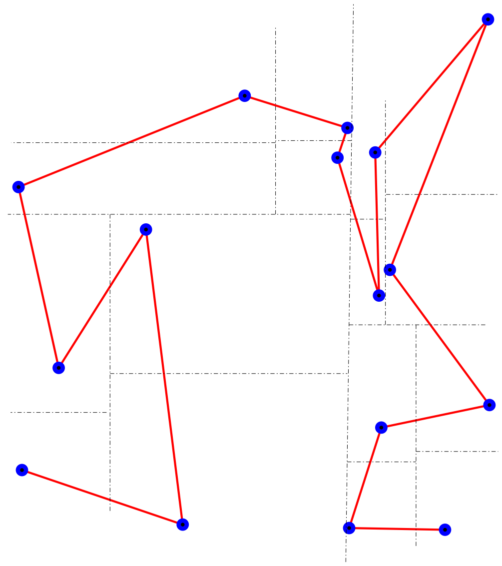


Figure 65.3: Hilbert sort with median policy

```
{
  std::vector<Point> v; v.reserve(4);
  v.push_back( Point(0.0,0.0)) ;
  v.push_back( Point(1.0,1.0)) ;
  v.push_back( Point(0.1,0.1)) ;
  v.push_back( Point(0.2,0.8)) ;

  std::cout << "Hilbert sort (middle policy)." << std::endl;
  CGAL::hilbert_sort (v.begin(), v.end(),
   CGAL::Hilbert_sort_middle_policy());
  std::cout<<v[0]<<"; "<<v[1]<<"; "<<v[2]<<"; "<<v[3]<<"; "<<std::endl;
  std::cout << "Hilbert sort (median policy)." << std::endl;
  CGAL::hilbert_sort (v.begin(), v.end(),
   CGAL::Hilbert_sort_median_policy());
  std::cout<<v[0]<<"; "<<v[1]<<"; "<<v[2]<<"; "<<v[3]<<"; "<<std::endl;
  return 0;
}
```

File: `examples/Spatial_sorting/hilbert_policies.cpp`

65.3 Spatial Sorting

Hilbert sort cannot be used directly before feeding a randomized algorithm. Thus, the trick is to organize the point set in random buckets of increasing sizes, Hilbert sort being used only inside a bucket.

It has been proved, in the context of Delaunay triangulation, that such an order provides enough randomness to combine the advantages of a random order and a space filling curve order [ACR03].

CGAL provides spatial sorting for points in 2D, 3D and higher dimensions, with the middle and the median policies for Hilbert sort in the buckets.

65.3.1 Basic Example

The following example shows that, on particular input, spatial sort runs much faster than a bad order or than Hilbert sort (below results with release mode compilation on a 1.8GHz processor).

```
$ ./small_example_delaunay_2
10000 points on a parabola
  Delaunay without spatial sort...        done in 6.33443 seconds.
  Delaunay with median hilbert sort...    done in 0.822975 seconds.
  Delaunay with median spatial sort...    done in 0.022415 seconds.
```

```
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/Delaunay_triangulation_2.h>
#include <CGAL/spatial_sort.h>
#include <CGAL/Timer.h>
#include <iostream>
#include <vector>
typedef CGAL::Exact_predicates_inexact_constructions_kernel K;
typedef CGAL::Delaunay_triangulation_2<K> DT;
void compute_delaunay(std::vector<K::Point_2>::iterator it,
                      std::vector<K::Point_2>::iterator e){
    DT dt;
    DT::Face_handle hint;
    for( ;it!=e; ++it)  hint = dt.insert(*it, hint)->face();
}
int main ()
{   int size = 1000;
    std::vector<K::Point_2> v;
    v.reserve(size);
    CGAL::Timer cost;
    std::cout <<size<< " points on a parabola" << std::endl;
    for (int i=0; i< size; ++i) {
      double x= -size +i;
      v.push_back( K::Point_2( x, x*x ));
    }
    cost.reset();cost.start();
    std::cout << "  Delaunay without spatial sort... "<< std::flush;
    compute_delaunay(v.begin(),v.end());cost.stop();
    std::cout << "done in "<<cost.time()<<" seconds." << std::endl;
    cost.reset();cost.start();
    std::cout << "  Delaunay with Hilbert sort...    " << std::flush;
    CGAL::hilbert_sort(v.begin(),v.end());
    compute_delaunay(v.begin(),v.end());cost.stop();
    std::cout << "done in "<<cost.time()<<" seconds." << std::endl;
    cost.reset();cost.start();
    std::cout << "  Delaunay with spatial sort...    " << std::flush;
    CGAL::spatial_sort(v.begin(),v.end());
    compute_delaunay(v.begin(),v.end());cost.stop();
    std::cout << "done in "<<cost.time()<<" seconds." << std::endl;
```

```
    return 0;
}
```

File: examples/Spatial_sorting/small_example_delaunay_2.cpp

65.3.2 Using Your Own Point Type

If you want to sort points of your own point type, you only have to provide functors that compare the *x* and *y* coordinates of your points. Note that in case you simply want to associate an extra information to your point you might consider the example of Section 65.3.3 as an alternative.

```
#include <CGAL/spatial_sort.h>

struct MyPoint {
  double x,y;
  int color;
  MyPoint()
    : x(0), y(0),color(0)
  {}

  MyPoint(double x, double y, int color=0)
    : x(x), y(y), color(color)
  {}
};


struct MyLessX {

  bool operator()(const MyPoint& p, const MyPoint& q) const
  {
    return p.x < q.x;
  }

};

struct MyLessY {

  bool operator()(const MyPoint& p, const MyPoint& q) const
  {
    return p.y < q.y;
  }

};

struct MySpatialSortingTraits {

  typedef MyPoint Point_2;

  typedef MyLessX Less_x_2;
  typedef MyLessY Less_y_2;

  Less_x_2 less_x_2_object() const
```

```
  {
    return Less_x_2();
  }

  Less_y_2 less_y_2_object() const
  {
    return Less_y_2();
  }
};

int main()
{
  std::vector< MyPoint > points;

  points.push_back(MyPoint(14,12, 3));
  points.push_back(MyPoint(1,2  , 0));
  points.push_back(MyPoint(414,2, 5));
  points.push_back(MyPoint(4,21 , 1));
  points.push_back(MyPoint(7,74 , 2));
  points.push_back(MyPoint(74,4 , 4));

  MySpatialSortingTraits sst;
  CGAL::spatial_sort(points.begin(), points.end(), sst);

  for (std::vector< MyPoint >::iterator
   it=points.begin();it!=points.end();++it)
    std::cout << it->color << " ";
  std::cout << "\n";

  std::cerr << "done" << std::endl;
  return 0;
}
```

File: examples/Spatial_sorting/myPoint.cpp

65.3.3 Sorting Arbitrary Types

The spatial sorting traits class provides a point type and functors for comparing, for example, the *x*-coordinates of two points. If you want to sort something else than just points, for example a sequence of tuples containing a point, or a sequence of indices in a vector of points, you need another level of indirection. We provide the spatial sorting traits class adapters which are templated by another spatial sorting traits class, and a property map. which allows to obtain a point from whatever you want to sort.

The following examples illustrate the usage of these traits class adapters.

Sorting Using Pairs of Points and Integers

In this example program, the sorted sequence of points is retrieved using a vector of pairs of points and integers.

```
#include <CGAL/Simple_cartesian.h>
```

```
#include <CGAL/spatial_sort.h>
#include <CGAL/Spatial_sort_traits_adapter_2.h>
#include <vector>

typedef CGAL::Simple_cartesian<double>       Kernel;
typedef Kernel::Point_2                      Point_2;
typedef std::pair<Point_2,int>               Point_with_info;
typedef std::vector< Point_with_info >       Data_vector;

//property map
struct First_of_pair{
  //classical typedefs
  typedef Point_with_info key_type;
  typedef Point_2 value_type;
  typedef const Point_2& reference;
  typedef boost::readable_property_map_tag category;
};
//get function for property map
First_of_pair::reference
get(const First_of_pair&, const First_of_pair::key_type& k) {
  return k.first;
}

typedef CGAL::Spatial_sort_traits_adapter_2<Kernel,First_of_pair>
    Search_traits_2;

int main()
{
  Data_vector points;
  points.push_back(std::make_pair(Point_2(14,12) , 3));
  points.push_back(std::make_pair(Point_2(1,2)   , 0));
  points.push_back(std::make_pair(Point_2(414,2) , 5));
  points.push_back(std::make_pair(Point_2(4,21)  , 1));
  points.push_back(std::make_pair(Point_2(7,74)  , 2));
  points.push_back(std::make_pair(Point_2(74,4)  , 4));

  Search_traits_2 traits;
  CGAL::spatial_sort(points.begin(), points.end(), traits);
  for (Data_vector::iterator it=points.begin();it!=points.end();++it)
    std::cout << it->second << " ";
  std::cout << "\n";

  std::cout << "done" << std::endl;

  return 0;
}
```

File: examples/Spatial_sorting/sp_sort_using_property_map_2.cpp

Sorting Using Indices of Points

In this example program, the sorted sequence of points is retrieved using the indices of the points in a vector of points.

```
#include <CGAL/Simple_cartesian.h>
#include <CGAL/spatial_sort.h>
#include <CGAL/Spatial_sort_traits_adapter_3.h>
#include <vector>
#include <boost/iterator/counting_iterator.hpp>

typedef CGAL::Simple_cartesian<double>                        Kernel;
typedef Kernel::Point_3                                       Point_3;
//using a pointer as a special property map type
typedef
  CGAL::Spatial_sort_traits_adapter_3<Kernel,Point_3*>   Search_traits_3;

int main()
{
  std::vector<Point_3> points;
  points.push_back(Point_3(1,3,11));
  points.push_back(Point_3(14,34,46));
  points.push_back(Point_3(414,34,4));
  points.push_back(Point_3(4,2,56));
  points.push_back(Point_3(744,4154,43));
  points.push_back(Point_3(74,44,1));

  std::vector<std::ptrdiff_t> indices;
  indices.reserve(points.size());

  std::copy(boost::counting_iterator<std::ptrdiff_t>(0),
            boost::counting_iterator<std::ptrdiff_t>(points.size()),
            std::back_inserter(indices));

  CGAL::spatial_sort(
   indices.begin(),indices.end(),Search_traits_3(&(points[0])) );

  for (std::vector<std::ptrdiff_t>::iterator
   it=indices.begin();it!=indices.end();++it)
    std::cout << points[*it] << "\n";

  std::cout << "done" << std::endl;

  return 0;
}
```

File: examples/Spatial_sorting/sp_sort_using_property_map_3.cpp

Sorting Using Indices of Pairs of Points and Integers

In this example program, the sorted sequence of points is retrieved using the indices of the points in a vector of pairs of points and integers.

```
#include <CGAL/Cartesian_d.h>
#include <CGAL/spatial_sort.h>
#include <CGAL/Spatial_sort_traits_adapter_d.h>
#include <boost/iterator/counting_iterator.hpp>
#include <vector>

typedef CGAL::Cartesian_d<double>          Kernel;
typedef Kernel::Point_d                    Point_d;
typedef std::pair<Point_d,int>             Point_with_info;
typedef std::vector< Point_with_info >     Data_vector;

//property map and get as friend
// to be allowed to use private member
class Vect_ppmap{
  const Data_vector& points;
public:
  //classical typedefs
  typedef Data_vector::size_type key_type;
  typedef Point_d value_type;
  typedef const value_type& reference;
  typedef boost::readable_property_map_tag category;

  Vect_ppmap(const Data_vector& points_):points(points_){}

  friend reference get(const Vect_ppmap& vmap, key_type i) {
    return vmap.points[i].first;
  }
};

typedef CGAL::Spatial_sort_traits_adapter_d<Kernel,Vect_ppmap>
   Search_traits_d;

int main()
{
  double coords[] ={ 1.0, 1.0, 1.0, 1.0,
                     2.0, 2.0, 2.0, 2.0 };

  Data_vector points;
  points.push_back(std::make_pair(Point_d(4,coords  ,coords+4) , 1));
  points.push_back(std::make_pair(Point_d(4,coords+4,coords+8) , 2));

  std::vector<Vect_ppmap::key_type> indices;
  indices.reserve(points.size());

  std::copy(
    boost::counting_iterator<Vect_ppmap::key_type>(0),
    boost::counting_iterator<Vect_ppmap::key_type>(points.size()),
    std::back_inserter(indices) );

  CGAL::spatial_sort(
    indices.begin(),
    indices.end(),
    Search_traits_d(Vect_ppmap(points)) );
```

```
  std::vector<Vect_ppmap::key_type>::iterator it=indices.begin();
  for (;it!=indices.end();++it)
    std::cout << points[*it].second << " ";
  std::cout << std::endl;

  std::cout << "done" << std::endl;

  return 0;
}
```

File: examples/Spatial_sorting/sp_sort_using_property_map_d.cpp

Spatial Sorting Reference Manual

Christophe Delage and Olivier Devillers

Functions

Function Objects

Traits classes

Concepts

Utilities

65.4 Alphabetical List of Reference Pages

CGAL::spatial_sort

Function

Definition

The function *spatial_sort* sorts an iterator range of points in a way that improves space locality. Two points close in the order will be close geometrically, and two points close geometrically will have a high probability of being close in the order.

#include <CGAL/spatial_sort.h>

```
template <class RandomAccessIterator, class Traits, class PolicyTag>
void              spatial_sort( RandomAccessIterator begin,
                                RandomAccessIterator end,
                                Traits traits = Default_traits,
                                PolicyTag policy = Default_policy,
                                std::ptrdiff_t threshold_hilbert=default,
                                std::ptrdiff_t threshold_multiscale=default,
                                double ratio=default)
```

sorts the range [*begin*,*end*) in place.

The default traits class *Default_traits* is the kernel in which the type *RandomAccessIterator::value_type* is defined.

The default policy is *Hilbert_sort_median_policy()* and the other option is *Hilbert_sort_middle_policy()*.

The default values for the thresholds and the ratio depends on the dimension.

Requirements

1. *RandomAccessIterator::value_type* is convertible to *Traits::Point_2*, *Traits::Point_3*, or *Traits::Point_d*.
2. *Traits* is a model for concept *SpatialSortingTraits_2*, *SpatialSortingTraits_3*, or *SpatialSortingTraits_d*.

Implementation

Creates an instance of *Multiscale_sort<Hilbert_sort>* where *Hilbert_sort* is an Hilbert sorting object, and calls its *operator()*.

The *threshold_hilbert* is the minimal size of a point set to be subdivided recursively during Hilbert sorting, otherwise random order is used. The *threshold_multiscale* value is the minimal size for a sample to call Hilbert sort, otherwise random order is used. The *ratio* value is used to split the original set in two subsets, spatial sort is applied on the first subset of size *ratio* times the original size of the set, Hilbert sort is applied on the second subset.

CGAL::hilbert_sort

Definition

The function *hilbert_sort* sorts an iterator range of points along a Hilbert curve.

#include <CGAL/hilbert_sort.h>

template <class RandomAccessIterator, class Traits, class PolicyTag>
void hilbert_sort(RandomAccessIterator begin,
RandomAccessIterator end,
Traits traits = Default_traits,
PolicyTag policy = Default_policy)

sorts the range [*begin*,*end*) in place.

The default traits class *Default_traits* is the kernel in which the type *RandomAccessIterator::value_type* is defined. The default policy is *Hilbert_sort_median_policy()* and the other option is *Hilbert_sort_middle_policy()*.

Requirements

1. *RandomAccessIterator::value_type* is convertible to *Traits::Point_2*, *Traits::Point_3*, or *Traits::Point_d*.
2. *Traits* is a model for concept *SpatialSortingTraits_2*, *SpatialSortingTraits_3*, or *SpatialSortingTraits_d*.

Implementation

Creates an instance of *Hilbert_sort_2<Traits, PolicyTag>*, *Hilbert_sort_3<Traits, PolicyTag>*, or *Hilbert_sort_d<Traits, PolicyTag>* and calls its *operator()*.

SpatialSortingTraits_2

Concept

Definition

All 2D spatial sorting algorithms provided in CGAL are parameterized by a traits class *Traits*, which defines the primitives (objects and predicates) that the sorting algorithms use. SpatialSortingTraits_2 defines the complete set of primitives required in these functions and functors.

Types

SpatialSortingTraits_2:: Point_2 — The point type on which the sorting algorithms operate.

SpatialSortingTraits_2:: Compute_x_2

Functor object type returning the x coordinate of a *Point_2*. Must provide *FT operator()(Point_2 p)* where *FT* can be used as argument of *CGAL::to_double*.

SpatialSortingTraits_2:: Compute_y_2

Functor object type returning the y coordinate of a *Point_2*. Must provide *FT operator()(Point_2 p)* where *FT* can be used as argument of *CGAL::to_double*.

SpatialSortingTraits_2:: Less_x_2 — Binary predicate object type comparing *Point_2*s along the x coordinate. Must provide *bool operator()(Point_2 p, Point_2 q)* where *true* is returned iff $p_x < q_x$, where p_x and q_x denote x coordinate of point p and q, respectively.

SpatialSortingTraits_2:: Less_y_2 — Binary predicate object type comparing *Point_2*s along the y coordinate. Must provide *bool operator()(Point_2 p, Point_2 q)* where *true* is returned iff $p_y < q_y$, where p_y and q_y denote y coordinate of point p and q, respectively.

Creation

Only a copy constructor is required.

SpatialSortingTraits_2 traits(t);

Operations

The following member functions to create instances of the above predicate object types must exist.

Compute_x_2	*traits.compute_x_2_object()*
Compute_y_2	*traits.compute_y_2_object()*
Less_x_2	*traits.less_x_2_object()*
Less_y_2	*traits.less_y_2_object()*

Has Models

Any CGAL kernel.
CGAL::Spatial_sort_traits_adapter_2<Base_traits,PointPropertyMap>.

CGAL::Hilbert_sort_2<Traits, PolicyTag>

Functor
Class

Definition

The function object *Hilbert_sort_2<Traits, PolicyTag>* sorts iterator ranges of *Traits::Point_2* along a Hilbert curve by recursively subdividing at the median or the middle depending on the *PolicyTag*.

Requirements

Traits is a model for *SpatialSortingTraits*.

Creation

Hilbert_sort_2<Traits, PolicyTag> o(const Traits &traits = Traits());

constructs an instance with *traits* as traits class instance.

Operations

template <class RandomAccessIterator>
void *o.operator()(RandomAccessIterator begin, RandomAccessIterator end) const*

sorts the range [*begin*, *end*).
Requirement: *RandomAccessIterator::value_type* equals to *Traits::Point_2*.

SpatialSortingTraits_3

Definition

All 3D spatial sorting algorithms provided in CGAL are parameterized by a traits class *Traits*, which defines the primitives (objects and predicates) that the sorting algorithms use. SpatialSortingTraits_3 defines the complete set of primitives required in these functions and functors.

Types

SpatialSortingTraits_3:: Point_3 — The point type on which the sorting algorithms operate.

SpatialSortingTraits_3:: Compute_x_3

Functor object type returning the x coordinate of a *Point_3*. Must provide *FT operator()(Point_3 p)* where *FT* can be used as argument of *CGAL::to_double*.

SpatialSortingTraits_3:: Compute_y_3

Functor object type returning the y coordinate of a *Point_3*. Must provide *FT operator()(Point_3 p)* where *FT* can be used as argument of *CGAL::to_double*.

SpatialSortingTraits_3:: Compute_z_3

Functor object type returning the z coordinate of a *Point_3*. Must provide *FT operator()(Point_3 p)* where *FT* can be used as argument of *CGAL::to_double*.

SpatialSortingTraits_3:: Less_x_3 — Binary predicate object type comparing *Point_3*s along the x coordinate. Must provide *bool operator()(Point_3 p, Point_3 q)* where *true* is returned iff $p_x < q_x$, where p_x and q_x denote x coordinate of point p and q, respectively.

SpatialSortingTraits_3:: Less_y_3 — Binary predicate object type comparing *Point_3*s along the y coordinate. Must provide *bool operator()(Point_3 p, Point_3 q)* where *true* is returned iff $p_y < q_y$, where p_y and q_y denote y coordinate of point p and q, respectively.

SpatialSortingTraits_3:: Less_z_3 — Binary predicate object type comparing *Point_3*s along the z coordinate. Must provide *bool operator()(Point_3 p, Point_3 q)* where *true* is returned iff $p_z < q_z$, where p_z and q_z denote z coordinate of point p and q, respectively.

Creation

Only a copy constructor is required.

SpatialSortingTraits_3 traits(t);

Operations

The following member functions to create instances of the above predicate object types must exist.

Compute_x_3	*traits.compute_x_3_object()*
Compute_y_3	*traits.compute_y_3_object()*
Compute_z_3	*traits.compute_z_3_object()*
Less_x_3	*traits.less_x_3_object()*
Less_y_3	*traits.less_y_3_object()*
Less_z_3	*traits.less_z_3_object()*

Has Models

Any CGAL kernel.
CGAL::Spatial_sort_traits_adapter_3<Base_traits,PointPropertyMap>.

Functor
Class

CGAL::Hilbert_sort_3<Traits, PolicyTag>

Definition

The function object *Hilbert_sort_3<Traits, PolicyTag>* sorts iterator ranges of *Traits::Point_3* along a Hilbert curve by recursively subdividing at the median or the middle depending on the *PolicyTag*.

Requirements

Traits is a model for *SpatialSortingTraits_3*.

Creation

Hilbert_sort_3<Traits, PolicyTag> o(const Traits &traits = Traits());

constructs an instance with *traits* as traits class instance.

Operations

template <class RandomAccessIterator>
void *o.operator()(RandomAccessIterator begin, RandomAccessIterator end) const*

sorts the range [*begin*, *end*).
Requirement: *RandomAccessIterator::value_type* equals to *Traits::Point_3*.

SpatialSortingTraits_d

Definition

All higher dimensional spatial sorting algorithms provided in CGAL are parameterized by a traits class *Traits*, which defines the primitives (objects and predicates) that the sorting algorithms use. SpatialSortingTraits_d defines the complete set of primitives required in these functions and functors.

Types

SpatialSortingTraits_d:: Point_d — The point type on which the sorting algorithms operate.

SpatialSortingTraits_d:: Point_dimension_d — Functor object type returning the dimension of a *Point_d*. Must provide *int operator()(Point_d p)* returning the dimension of p.

SpatialSortingTraits_d:: Compute_coordinate_d

Functor object type returning the coordinates of a *Point_d*. Must provide *FT operator()(Point_d p, int i)* returning the ith coordinate of p. *FT* is a type that can be used as argument of *CGAL::to_double*.

SpatialSortingTraits_d:: Less_coordinate_d — Binary predicate object type comparing *Point_d*s along some coordinate. Must provide *bool operator()(Point_d p, Point_d q, int i)* where *true* is returned iff $p_i < q_i$, where p_i and q_i denote ith coordinate of point p and q, respectively.

Creation

Only a copy constructor is required.

SpatialSortingTraits_d traits(t);

Operations

The following member functions to create instances of the above predicate object types must exist.

Point_dimension_d — *traits.point_dimension_d_object()*

Compute_coordinate_d

traits.compute_coordinate_d_object()

Less_coordinate_d — *traits.less_coordinate_d_object()*

Has Models

Any CGAL d dimensional kernel.
CGAL::Spatial_sort_traits_adapter_d<Base_traits,PointPropertyMap>.

Functor Class

CGAL::Hilbert_sort_d<Traits, PolicyTag>

Definition

The function object *Hilbert_sort_d<Traits, PolicyTag>* sorts iterator ranges of *Traits::Point_d* along a Hilbert curve by recursively subdividing at the median or the middle depending on the *PolicyTag*.

Requirements

Traits is a model for *SpatialSortingTraits_d*.

Creation

Hilbert_sort_d<Traits, PolicyTag> o(const Traits &traits = Traits());

constructs an instance with *traits* as traits class instance.

Operations

template <class RandomAccessIterator>
void *o.operator()(RandomAccessIterator begin, RandomAccessIterator end) const*

sorts the range [*begin*, *end*).
Requirement: *RandomAccessIterator::value_type* equals to *Traits::Point_d*.

CGAL::Multiscale_sort<Sort>

Functor
Class

Definition

The class *Multiscale_sort<Sort>* represents a sorting algorithm adaptor. Given a range of n points:

1. it applies *Sort* on the last $(1-ratio) \times n$ points,
2. it recurses on the first $ratio \times n$ points, stopping when there are less than *threshold* points.

Creation

Multiscale_sort<Sort> o(const Sort &sort = Sort(), std::ptrdiff_t threshold = 1, double ratio = 0.5);

constructs an instance with *traits* as traits class instance.

Operations

template <class RandomAccessIterator>
void *o.operator()(RandomAccessIterator begin, RandomAccessIterator end) const*

sorts the range [*begin*, *end*).
Requirement: *Sort::operator()(RandomAccessIterator begin, RandomAccessIterator end)* is defined.

Class

CGAL::Median

Definition

Median is a tag class. It can be used to parameterize a strategy policy in order to specify the strategy for spatial sorting. *Hilbert_policy<Median>* can be passed to as parameter to *hilbert_sort* to choose the sorting policy.

#include <CGAL/Hilbert_policy_tags.h>

Is Model for the Concepts

DefaultConstructible, CopyConstructible

See Also

Middle
Hilbert_policy
Hilbert_sort__median_policy
Hilbert_sort__middle_policy

CGAL::Middle

Definition

Middle is a tag class. It can be used to parameterize a strategy policy in order to specify the strategy for spatial sorting. *Hilbert_policy<Middle>* can be passed to as parameter to *hilbert_sort* to choose the sorting policy.

#include <CGAL/Hilbert_policy_tags.h>

Is Model for the Concepts

DefaultConstructible, CopyConstructible

See Also

Median
Hilbert_policy
Hilbert_sort__median_policy
Hilbert_sort__middle_policy

Class

CGAL::Hilbert_policy<Tag>

Definition

Hilbert_policy<Tag> is a policy class which can be used to parameterize a strategy policy in order to specify the strategy for spatial sorting. *Hilbert_policy<Median>* or *Hilbert_policy<Middle>* can be passed to as parameter to *hilbert_sort* to choose the sorting policy.

#*include <CGAL/Hilbert_policy_tags.h>*

Parameters

Tag can only be either *Median* or *Middle* currently.

Is Model for the Concepts

DefaultConstructible, CopyConstructible

See Also

Median
Middle
Hilbert_sort__median_policy
Hilbert_sort__middle_policy

CGAL::Hilbert_sort_median_policy

Definition

A typedef to *Hilbert_policy<Median>*.

#include <CGAL/Hilbert_policy_tags.h>

See Also

Median
Middle
Hilbert_policy
Hilbert_sort__middle_policy

Class

CGAL::Hilbert_sort_middle_policy

Definition

A typedef to *Hilbert_policy<Middle>*.

#include <CGAL/Hilbert_policy_tags.h>

See Also

Median
Middle
Hilbert_sort__median_policy
Hilbert_policy

CGAL::Spatial_sort_traits_adapter_2<Base_traits,PointPropertyMap>

Class

Definition

Given a property map associating a key to a point, the class *Spatial_sort_traits_adapter_2<Base_traits,PointPropertyMap>* induces a spatial reorder of the keys instead of the points, the comparisons being done on the associated points. In other words, the traits provides to a spatial sort algorithm a point type which is a key, while the actual point type is *Base_traits::Point_2*.

Requirements

Base_traits is a model for *SpatialSortingTraits_2*. *PointPropertyMap* is a model of boost::ReadablePropertyMap with *Base_traits::Point_2* as *value_type*.

Inherits From

Base_traits

Is Model for the Concepts

SpatialSortingTraits_2

Types

boost::property_traits<PointPropertyMap>::key_type

Point_2;

Creation

Spatial_sort_traits_adapter_2<Base_traits,PointPropertyMap> o(Base_traits base=Base_traits());

Spatial_sort_traits_adapter_2<Base_traits,PointPropertyMap> o(const PointPropertyMap& ppmap, Base_traits base=Base_traits())

Operations

const PointPropertyMap&

o.point_property_map() const

Returns a const reference to the point property map.

Class

CGAL::Spatial_sort_traits_adapter_3<Base_traits,PointPropertyMap>

Definition

Given a property map associating a key to a point, the class *Spatial_sort_traits_adapter_3<Base_traits,PointPropertyMap>* induces a spatial reorder of the keys instead of the points, the comparisons being done on the associated points. In other words, the traits provides to a spatial sort algorithm a point type which is a key, while the actual point type is *Base_traits::Point_3*.

Requirements

Base_traits is a model for *SpatialSortingTraits_3*. *PointPropertyMap* is a model of boost::ReadablePropertyMap with *Base_traits::Point_3* as *value_type*.

Inherits From

Base_traits

Is Model for the Concepts

SpatialSortingTraits_3

Types

boost::property_traits<PointPropertyMap>::key_type

Point_3;

Creation

Spatial_sort_traits_adapter_3<Base_traits,PointPropertyMap> o(Base_traits base=Base_traits());

Spatial_sort_traits_adapter_3<Base_traits,PointPropertyMap> o(const PointPropertyMap& ppmap, Base_traits base=Base_traits())

Operations

const PointPropertyMap&

o.point_property_map() const

Returns a const reference to the point property map.

CGAL::Spatial_sort_traits_adapter_d<Base_traits,PointPropertyMap>

Class

Definition

Given a property map associating a key to a point, the class *Spatial_sort_traits_adapter_d<Base_traits,PointPropertyMap>* induces a spatial reorder of the keys instead of the points, the comparisons being done on the associated points. In other words, the traits provides to a spatial sort algorithm a point type which is a key, while the actual point type is *Base_traits::Point_d*.

Requirements

Base_traits is a model for *SpatialSortingTraits_d*. *PointPropertyMap* is a model of boost::ReadablePropertyMap with *Base_traits::Point_d* as *value_type*.

Inherits From

Base_traits

Is Model for the Concepts

SpatialSortingTraits_d

Types

boost::property_traits<PointPropertyMap>::key_type

Point_d;

Creation

Spatial_sort_traits_adapter_d<Base_traits,PointPropertyMap> o(Base_traits base=Base_traits());

Spatial_sort_traits_adapter_d<Base_traits,PointPropertyMap> o(const PointPropertyMap& ppmap, Base_traits base=Base_traits())

Operations

const PointPropertyMap&

o.point_property_map() const

Returns a const reference to the point property map.

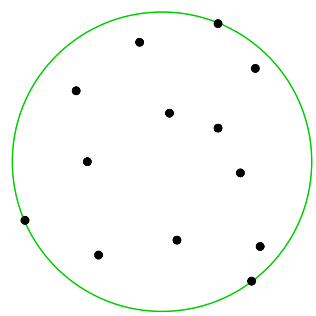
Part XIV

Geometric Optimization

Chapter 66

Bounding Volumes

Kaspar Fischer, Bernd Gärtner, Thomas Herrmann, Michael Hoffmann, and Sven Schönherr



This chapter describes algorithms which for a given point set compute the "best" circumscribing object from a specific class. If the class consists of all spheres in d-dimensional Euclidean space and "best" is defined as having smallest radius, then we obtain the smallest enclosing sphere problem already mentioned above.

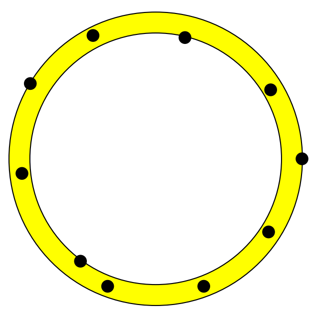
In the following example a smallest enclosing circle (*CGAL::Min_circle_2<Traits>*) is constructed from points on a line and written to standard output. The example shows that it is advisable to switch on random shuffling in order to deal with a 'bad' order of the input points.

```
#include <CGAL/Exact_predicates_exact_constructions_kernel.h>
#include <CGAL/Min_circle_2.h>
#include <CGAL/Min_circle_2_traits_2.h>
#include <iostream>

// typedefs
typedef  CGAL::Exact_predicates_exact_constructions_kernel K;
typedef  CGAL::Min_circle_2_traits_2<K>  Traits;
typedef  CGAL::Min_circle_2<Traits>      Min_circle;

typedef  K::Point_2                      Point;

int
main( int, char**)
{
    const int n = 100;
    Point P[n];
```

```
    for ( int i = 0; i < n; ++i)
        P[ i] = Point( (i%2 == 0 ? i : -i), 0);
    // (0,0), (-1,0), (2,0), (-3,0), ...

    Min_circle  mc1( P, P+n, false);    // very slow
    Min_circle  mc2( P, P+n, true);     // fast

    CGAL::set_pretty_mode( std::cout);
    std::cout << mc2;

    return 0;
}
```

File: examples/Min_circle_2/min_circle_2.cpp

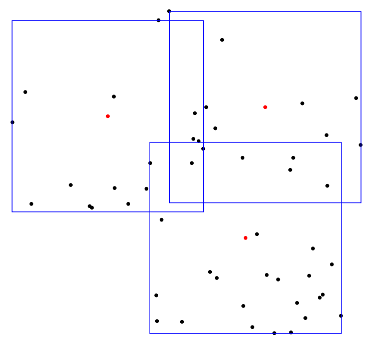
Other classes for which we provide solutions are ellipses (*CGAL::Min_ellipse_2<Traits>*), rectangles (*CGAL::min_rectangle_2*), parallelograms (*CGAL::min_parallelogram_2*) and strips (*CGAL::min_strip_2*) in the plane, with appropriate optimality criteria. For arbitrary dimensions we provide smallest enclosing spheres for points (*CGAL::Min_sphere_d<Traits>*) and spheres for spheres (*CGAL::Min_sphere_of_spheres_d<Traits>*), smallest enclosing annuli (*CGAL::Min_annulus_d<Traits>*), and approximate minimum-volume enclosing ellipsoid with user-specified approximation ratio (*CGAL::Approximate_min_ellipsoid_d<Traits>*).



Bounding volumes can be used to obtain simple approximations of complicated objects. For example, consider the problem of deciding whether two moving polygons currently intersect. An obvious solution is to discretize time and perform a full intersection test for any time step. If the polygons are far apart most of the time, this is unnecessary. Instead, simple bounding volumes (for examples, circles) are computed for both polygons at their initial positions. At subsequent time steps, an intersection test between the moving bounding circles replaces the actual intersection test; only if the circles do intersect, the expensive intersection test between the polygons is performed. In practice, bounding volume hierarchies are often used on top of simple bounding volumes to approximate complicated objects more accurately.

Bounding volumes are also frequently applied to extract geometric properties of objects. For example, the smallest enclosing annulus of a point set can be used to test whether a set of points is approximately cospherical. Here, the width of the annulus (or its area, or still another criterion that we use) is a good measure for this property. The largest area triangle is for example used in heuristics for matching archaeological aerial photographs. Largest perimeter triangles are used in scoring cross country soaring flights, where the goal is basically to fly as far as possible, but still return to the departure airfield. To score simply based on the total distance flown is not a good measure, since circling in thermals allows to increase it easily.

Bounding volumes also define geometric “center points” of objects. For example, if two objects are to be matched (approximately), one approch is to first apply the translation that maps the centers of their smallest enclosing spheres onto each other. Simpler centers are possible, of course (center of gravity, center of bounding box), but more advanced bounding volumes might give better results in some cases. It can also make sense to consider several center points instead of just one. For example, we provide algorithms to cover a planar point set with between two and four minimal boxes (*CGAL::rectangular_p_center_2*). Below is an example covering with three boxes; the center points are shown in red.



Bounded Volumes Reference Manual

Kaspar Fischer, Bernd Gärtner, Thomas Herrmann, Michael Hoffmann, and Sven Schönherr

This chapter describes concepts, classes, and functions for bounding areas and volumes.

These are Smallest enclosing circle and ellipse (2D), smallest enclosing rectangle, parallelogram, and strip (2D), rectangular p-center (2D), smallest enclosing sphere and annulus (dD), approximate minimum-volume enclosing ellipsoid with user-specified approximation ratio (dD).

Assertions

The optimization code uses infix *OPTIMISATION* in the assertions, e.g. defining the compiler flag *CGAL_OPTIMISATION_NO_PRECONDITIONS* switches precondition checking off, cf. Section 2.8.

66.1 Classified References Pages

Bounding Areas and Volumes

66.2 Alphabetical List of Reference Pages

CGAL::Min_circle_2<Traits>

Definition

An object of the class *Min_circle_2<Traits>* is the unique circle of smallest area enclosing a finite (multi)set of points in two-dimensional Euclidean space $\mathbb{E}^2$. For a point set P we denote by $mc(P)$ the smallest circle that contains all points of P. Note that $mc(P)$ can be degenerate, i.e. $mc(P) = \emptyset$ if $P = \emptyset$ and $mc(P) = \{p\}$ if $P = \{p\}$.

An inclusion-minimal subset S of P with $mc(S) = mc(P)$ is called a *support set*, the points in S are the *support points*. A support set has size at most three, and all its points lie on the boundary of $mc(P)$. In general, neither the support set nor its size are necessarily unique.

The underlying algorithm can cope with all kinds of input, e.g. P may be empty or points may occur more than once. The algorithm computes a support set S which remains fixed until the next insert or clear operation.

Please note: This class is (almost) obsolete. The class *CGAL::Min_sphere_of_spheres_d<Traits>* solves a more general problem and is faster then *Min_circle_2<Traits>* even if used only for points in two dimensions as input. Most importantly, *CGAL::Min_sphere_of_spheres_d<Traits>* has a specialized implementation for floating-point arithmetic which ensures correct results in a large number of cases (including highly degenerate ones). In contrast, *Min_circle_2<Traits>* is not tuned for floating-point computations. The only advantage of *Min_circle_2<Traits>* over *CGAL::Min_sphere_of_spheres_d<Traits>* is that the former can deal with points in homogeneous coordinates, in which case the algorithm is division-free. Thus, *Min_circle_2<Traits>* might still be an option in case your input number type cannot (efficiently) divide.

#include <CGAL/Min_circle_2.h>

Requirements

The template parameter *Traits* is a model for *MinCircle2Traits*.

We provide the model *CGAL::Min_circle_2_traits_2* using the two-dimensional CGAL kernel.

Types

Min_circle_2<Traits>:: Point typedef to *Traits::Point*.

Min_circle_2<Traits>:: Circle typedef to *Traits::Circle*.

Min_circle_2<Traits>:: Point_iterator

non-mutable model of the STL concept *BidirectionalIterator* with value type *Point*. Used to access the points of the smallest enclosing circle.

Min_circle_2<Traits>:: Support_point_iterator

non-mutable model of the STL concept *RandomAccessIterator* with value type *Point*. Used to access the support points of the smallest enclosing circle.

Creation

A *Min_circle_2<Traits>* object can be created from an arbitrary point set P and by specialized construction methods expecting no, one, two or three points as arguments. The latter methods can be useful for reconstructing $mc(P)$ from a given support set S of P.

template < class InputIterator >
Min_circle_2<Traits> min_circle(InputIterator first, InputIterator last, bool randomize, Random& random = CGAL::default_random, Traits traits = Traits())

initializes *min_circle* to $mc(P)$ with P being the set of points in the range [*first*,*last*). If *randomize* is *true*, a random permutation of P is computed in advance, using the random numbers generator *random*. Usually, this will not be necessary, however, the algorithm's efficiency depends on the order in which the points are processed, and a bad order might lead to extremely poor performance (see example below).
Requirement: The value type of *first* and *last* is *Point*.

Min_circle_2<Traits> min_circle(Traits traits = Traits());

initializes *min_circle* to $mc(\emptyset)$, the empty set.
Postcondition: *min_circle.is_empty() = true*.

Min_circle_2<Traits> min_circle(Point p, Traits traits = Traits());

initializes *min_circle* to $mc(\{p\})$, the set $\{p\}$.
Postcondition: *min_circle.is_degenerate() = true*.

Min_circle_2<Traits> min_circle(Point p1, Point p2, Traits traits = Traits());

initializes *min_circle* to $mc(\{p1,p2\})$, the circle with diameter equal to the segment connecting $p1$ and $p2$.

Min_circle_2<Traits> min_circle(Point p1, Point p2, Point p3, Traits traits = Traits());

initializes *min_circle* to $mc(\{p1,p2,p3\})$.

Access Functions

int *min_circle.number_of_points() const*

returns the number of points of *min_circle*, i.e. $|P|$.

int *min_circle.number_of_support_points() const*

returns the number of support points of *min_circle*, i.e. $|S|$.

Point_iterator
min_circle.points_begin() const
returns an iterator referring to the first point of *min_circle*.

Point_iterator
min_circle.points_end() const returns the corresponding past-the-end iterator.

Support_point_iterator
min_circle.support_points_begin() const
returns an iterator referring to the first support point of *min_circle*.

Support_point_iterator
min_circle.support_points_end() const
returns the corresponding past-the-end iterator.

Point *min_circle.support_point(int i) const*

returns the i-th support point of *min_circle*. Between two modifying operations (see below) any call to *min_circle.support_point(i)* with the same i returns the same point.
Precondition: $0 \leq i <$ *min_circle.number_of_support_points()*.

Circle *min_circle.circle() const* returns the current circle of *min_circle*.

Predicates

By definition, an empty *Min_circle_2<Traits>* has no boundary and no bounded side, i.e. its unbounded side equals the whole space $\mathbb{E}^2$.

CGAL::Bounded_side

min_circle.bounded_side(Point p) const

returns *CGAL::ON_BOUNDED_SIDE*, *CGAL::ON_BOUNDARY*, or *CGAL::ON_UNBOUNDED_SIDE* iff p lies properly inside, on the boundary of, or properly outside of *min_circle*, resp.

bool *min_circle.has_on_bounded_side(Point p) const*

returns *true*, iff p lies properly inside *min_circle*.

bool *min_circle.has_on_boundary(Point p) const*

returns *true*, iff p lies on the boundary of *min_circle*.

bool *min_circle.has_on_unbounded_side(Point p) const*

returns *true*, iff p lies properly outside of *min_circle*.

bool *min_circle.is_empty() const* returns *true*, iff *min_circle* is empty (this implies degeneracy).

bool *min_circle.is_degenerate() const*

returns *true*, iff *min_circle* is degenerate, i.e. if *min_circle* is empty or equal to a single point, equivalently if the number of support points is less than 2.

Modifiers

New points can be added to an existing *min_circle*, allowing to build $mc(P)$ incrementally, e.g. if P is not known in advance. Compared to the direct creation of $mc(P)$, this is not much slower, because the construction method is incremental itself.

void *min_circle.insert(Point p)* inserts *p* into *min_circle* and recomputes the smallest enclosing circle.

template < class InputIterator >
void *min_circle.insert(InputIterator first, InputIterator last)*

inserts the points in the range [*first*,*last*) into *min_circle* and recomputes the smallest enclosing circle by calling *insert(p)* for each point *p* in [*first*,*last*).
Requirement: The value type of *first* and *last* is *Point*.

void *min_circle.clear()* deletes all points in *min_circle* and sets *min_circle* to the empty set.
Postcondition: *min_circle.is_empty() = true*.

Validity Check

An object *min_circle* is valid, iff

- *min_circle* contains all points of its defining set *P*,
- *min_circle* is the smallest circle spanned by its support set *S*, and
- *S* is minimal, i.e. no support point is redundant.

bool *min_circle.is_valid(bool verbose = false, int level = 0) const*

returns *true*, iff *min_circle* is valid. If *verbose* is *true*, some messages concerning the performed checks are written to standard error stream. The second parameter *level* is not used, we provide it only for consistency with interfaces of other classes.

Miscellaneous

const Traits&

min_circle.traits() const returns a const reference to the traits class object.

I/O

std::ostream&

std::ostream& os << min_circle

writes *min_circle* to output stream *os*.
Requirement: The output operator is defined for *Point* (and for *Circle*, if pretty printing is used).

std::istream&

std::istream& is >> min_circle&

reads *min_circle* from input stream *is*.
Requirement: The input operator is defined for *Point*.

See Also

CGAL::Min_ellipse_2<Traits> . . . . page 3812
CGAL::Min_sphere_d<Traits> . . . . page 3846
CGAL::Min_sphere_of_spheres_d<Traits> . . . . page 3867
CGAL::Min_circle_2_traits_2<K> . . . . page 3808
MinCircle2Traits . . . . page 3809

Implementation

We implement the incremental algorithm of Welzl, with move-to-front heuristic [Wel91]. The whole implementation is described in [GS98a].

If randomization is chosen, the creation time is almost always linear in the number of points. Access functions and predicates take constant time, inserting a point might take up to linear time, but substantially less than computing the new smallest enclosing circle from scratch. The clear operation and the check for validity each takes linear time.

Example

To illustrate the creation of *Min_circle_2<Traits>* and to show that randomization can be useful in certain cases, we give an example.

```
#include <CGAL/Exact_predicates_exact_constructions_kernel.h>
#include <CGAL/Min_circle_2.h>
#include <CGAL/Min_circle_2_traits_2.h>
#include <iostream>

// typedefs
typedef  CGAL::Exact_predicates_exact_constructions_kernel K;
typedef  CGAL::Min_circle_2_traits_2<K>   Traits;
typedef  CGAL::Min_circle_2<Traits>       Min_circle;
```

```
typedef  K::Point_2                      Point;

int
main( int, char**)
{
    const int n = 100;
    Point P[n];

    for ( int i = 0; i < n; ++i)
        P[ i] = Point( (i%2 == 0 ? i : -i), 0);
    // (0,0), (-1,0), (2,0), (-3,0), ...

    Min_circle  mc1( P, P+n, false);    // very slow
    Min_circle  mc2( P, P+n, true);     // fast

    CGAL::set_pretty_mode( std::cout);
    std::cout << mc2;

    return 0;
}
```

File: examples/Min_circle_2/min_circle_2.cpp

Class

CGAL::Min_circle_2_traits_2<K>

Definition

The class *Min_circle_2_traits_2<K>* is a traits class for *Min_circle_2<Traits>* using the two-dimensional CGAL kernel.

#include <CGAL/Min_circle_2_traits_2.h>

Requirements

The template parameter *K* is a model for *Kernel*.

Is Model for the Concepts

MinCircle2Traits . . . . page 3809

Types

Min_circle_2_traits_2<K>:: Point typedef to *K::Point_2*.

Min_circle_2_traits_2<K>:: Circle internal type.

Creation

Min_circle_2_traits_2<K> traits; default constructor.

Min_circle_2_traits_2<K> traits(Min_circle_2_traits_2<K>);

copy constructor.

Operations

CGAL::Orientation *traits.orientation(Point p, Point q, Point r) const*

returns *CGAL::orientation(p, q, r)*.

See Also

CGAL::Min_circle_2<Traits> . . . . page 3802
MinCircle2Traits . . . . page 3809

MinCircle2Traits

Concept

Definition

This concept defines the requirements for traits classes of *CGAL::Min_circle_2<Traits>*.

Types

MinCircle2Traits:: Point	The point type must provide default and copy constructor, assignment and equality test.
MinCircle2Traits:: Circle	The circle type must fulfill the requirements listed below in the next section.

Variables

Circle	*circle;*	The current circle. This variable is maintained by the algorithm, the user should neither access nor modify it directly.

Creation

Only default and copy constructor are required.

MinCircle2Traits traits;

MinCircle2Traits traits(MinCircle2Traits);

Operations

The following predicate is only needed, if the member function *is_valid* of *Min_circle_2* is used.

CGAL::Orientation *traits.orientation(Point p, Point q, Point r) const*

returns constants *CGAL::LEFT_TURN*, *CGAL::COLLINEAR*, or *CGAL::RIGHT_TURN* iff r lies properly to the left of, on, or properly to the right of the oriented line through p and q, resp.

Has Models

CGAL::Min_circle_2_traits_2<K> . . . . page 3808

See Also

CGAL::Min_circle_2<Traits> . . . . page 3802

Circle Type (*Circle*)

Definition

An object of the class *Circle* is a circle in two-dimensional Euclidean plane $\mathbb{E}^2$. Its boundary splits the plane into a bounded and an unbounded side. By definition, an empty *Circle* has no boundary and no bounded side, i.e. its unbounded side equals the whole plane $\mathbb{E}^2$. A *Circle* containing exactly one point p has no bounded side, its boundary is $\{p\}$, and its unbounded side equals $\mathbb{E}^2 \setminus \{p\}$.

Types

Circle:: Point	Point type.

The following type is only needed, if the member function *is_valid* of *Min_circle_2* is used.

Circle:: Distance	Distance type. The function *squared_radius* (see below) returns an object of this type.

Creation

void	*circle.set()*	sets *circle* to the empty circle.
void	*circle.set(Point p)*	sets *circle* to the circle containing exactly $\{p\}$.
void	*circle.set(Point p, Point q)*	sets *circle* to the circle with diameter equal to the segment connecting p and q. The algorithm guarantees that *set* is never called with two equal points.
void	*circle.set(Point p, Point q, Point r)*	sets *circle* to the circle through p,q,r. The algorithm guarantees that *set* is never called with three collinear points.

Predicates

bool	*circle.has_on_unbounded_side(Point p) const*	returns *true*, iff p lies properly outside of *circle*.

Each of the following predicates is only needed, if the corresponding predicate of *Min_circle_2* is used.

CGAL::Bounded_side	*circle.bounded_side(Point p) const*	returns *CGAL::ON_BOUNDED_SIDE*, *CGAL::ON_BOUNDARY*, or *CGAL::ON_UNBOUNDED_SIDE* iff p lies properly inside, on the boundary, or properly outside of *circle*, resp.

bool *circle.has_on_bounded_side(Point p) const*

returns *true*, iff *p* lies properly inside *circle*.

bool *circle.has_on_boundary(Point p) const*

returns *true*, iff *p* lies on the boundary of *circle*.

bool *circle.is_empty() const* returns *true*, iff *circle* is empty (this implies degeneracy).

bool *circle.is_degenerate() const* returns *true*, iff *circle* is degenerate, i.e. if *circle* is empty or equal to a single point.

Additional Operations for Checking

The following operations are only needed, if the member function *is_valid* of *Min_circle_2* is used.

bool *circle == circle2* returns *true*, iff *circle* and *circle2* are equal.

Point *circle.center() const* returns the center of *circle*.

Distance *circle.squared_radius() const*

returns the squared radius of *circle*.

I/O

The following I/O operator is only needed, if the corresponding I/O operator of *Min_circle_2* is used.

std::ostream&

std::ostream& os << circle writes *circle* to output stream *os*.

Class

CGAL::Min_ellipse_2<Traits>

Definition

An object of the class *Min_ellipse_2<Traits>* is the unique ellipse of smallest area enclosing a finite (multi)set of points in two-dimensional euclidean space $\mathbb{E}^2$. For a point set P we denote by $me(P)$ the smallest ellipse that contains all points of P. Note that $me(P)$ can be degenerate, i.e. $me(P) = \emptyset$ if $P = \emptyset$, $me(P) = \{p\}$ if $P = \{p\}$, and $me(P) = \{(1-\lambda)p + \lambda q \mid 0 \leq \lambda \leq 1\}$ if $P = \{p,q\}$.

An inclusion-minimal subset S of P with $me(S) = me(P)$ is called a *support set*, the points in S are the *support points*. A support set has size at most five, and all its points lie on the boundary of $me(P)$. In general, neither the support set nor its size are necessarily unique.

The underlying algorithm can cope with all kinds of input, e.g. P may be empty or points may occur more than once. The algorithm computes a support set S which remains fixed until the next insert or clear operation.

#include <CGAL/Min_ellipse_2.h>

Requirements

The template parameter *Traits* is a model for *MinEllipse2Traits*.

We provide the model *CGAL::Min_ellipse_2_traits_2<K>* using the two-dimensional CGAL kernel.

Types

Min_ellipse_2<Traits>:: Point — Typedef to *Traits::Point*.

Min_ellipse_2<Traits>:: Ellipse — Typedef to *Traits::Ellipse*. If you are using the predefined traits class *CGAL::Min_ellipse_2_traits_2<K>*, you can access the coefficients of the ellipse, see the documentation of *CGAL::Min_ellipse_2_traits_2<K>* and the example below.

Min_ellipse_2<Traits>:: Point_iterator

Non-mutable model of the STL concept *BidirectionalIterator* with value type *Point*. Used to access the points of the smallest enclosing ellipse.

Min_ellipse_2<Traits>:: Support_point_iterator

Non-mutable model of the STL concept *RandomAccessIterator* with value type *Point*. Used to access the support points of the smallest enclosing ellipse.

Creation

A *Min_ellipse_2<Traits>* object can be created from an arbitrary point set P and by specialized construction methods expecting no, one, two, three, four or five points as arguments. The latter methods can be useful for reconstructing $me(P)$ from a given support set S of P.

template < class InputIterator >
Min_ellipse_2<Traits> min_ellipse(InputIterator first, InputIterator last, bool randomize, Random& random = default_random, Traits traits = Traits())

initializes *min_ellipse* to $me(P)$ with P being the set of points in the range [*first*,*last*). If *randomize* is *true*, a random permutation of P is computed in advance, using the random numbers generator *random*. Usually, this will not be necessary, however, the algorithm's efficiency depends on the order in which the points are processed, and a bad order might lead to extremely poor performance (see example below).
Requirement: The value type of *first* and *last* is *Point*.

Min_ellipse_2<Traits> min_ellipse(Traits traits = Traits());

creates a variable *min_ellipse* of type *Min_ellipse_2<Traits>*. It is initialized to $me(\emptyset)$, the empty set.
Postcondition: *min_ellipse.is_empty() = true*.

Min_ellipse_2<Traits> min_ellipse(Point p, Traits traits = Traits());

initializes *min_ellipse* to $me(\{p\})$, the set $\{p\}$.
Postcondition: *min_ellipse.is_degenerate() = true*.

Min_ellipse_2<Traits> min_ellipse(Point p, Point q, Traits traits = Traits());

initializes *min_ellipse* to $me(\{p,q\})$,
the set $\{(1-\lambda)p+\lambda q \mid 0 \leq \lambda \leq 1\}$.
Postcondition: *min_ellipse.is_degenerate() = true*.

Min_ellipse_2<Traits> min_ellipse(Point p1, Point p2, Point p3, Traits traits = Traits());

initializes *min_ellipse* to $me(\{p1,p2,p3\})$.

Min_ellipse_2<Traits> min_ellipse(Point p1, Point p2, Point p3, Point p4, Traits traits = Traits());

initializes *min_ellipse* to $me(\{p1,p2,p3,p4\})$.

Min_ellipse_2<Traits> min_ellipse(Point p1, Point p2, Point p3, Point p4, Point p5, Traits traits = Traits());

initializes *min_ellipse* to $me(\{p1,p2,p3,p4,p5\})$.

Access Functions

int *min_ellipse.number_of_points() const*

returns the number of points of *min_ellipse*, i.e. $|P|$.

int *min_ellipse.number_of_support_points() const*

returns the number of support points of *min_ellipse*, i.e. $|S|$.

Point_iterator
min_ellipse.points_begin() const
returns an iterator referring to the first point of *min_ellipse*.

Point_iterator
min_ellipse.points_end() const returns the corresponding past-the-end iterator.

Support_point_iterator
min_ellipse.support_points_begin() const
returns an iterator referring to the first support point of *min_ellipse*.

Support_point_iterator
min_ellipse.support_points_end() const
returns the corresponding past-the-end iterator.

Point *min_ellipse.support_point(int i) const*

returns the i-th support point of *min_ellipse*. Between two modifying operations (see below) any call to *min_ellipse.support_point(i)* with the same i returns the same point.
Precondition: $0 \leq i <$ *min_ellipse.number_of_support_points()*.

Ellipse *min_ellipse.ellipse() const* returns the current ellipse of *min_ellipse*.

Predicates

By definition, an empty *Min_ellipse_2<Traits>* has no boundary and no bounded side, i.e. its unbounded side equals the whole space $\mathbb{E}^2$.

CGAL::Bounded_side

min_ellipse.bounded_side(Point p) const

returns *CGAL::ON_BOUNDED_SIDE*, *CGAL::ON_BOUNDARY*, or *CGAL::ON_UNBOUNDED_SIDE* iff p lies properly inside, on the boundary of, or properly outside of *min_ellipse*, resp.

bool *min_ellipse.has_on_bounded_side(Point p) const*

returns *true*, iff p lies properly inside *min_ellipse*.

bool *min_ellipse.has_on_boundary(Point p) const*

returns *true*, iff *p* lies on the boundary of *min_ellipse*.

bool *min_ellipse.has_on_unbounded_side(Point p) const*

returns *true*, iff *p* lies properly outside of *min_ellipse*.

bool *min_ellipse.is_empty() const* returns *true*, iff *min_ellipse* is empty (this implies degeneracy).

bool *min_ellipse.is_degenerate() const*

returns *true*, iff *min_ellipse* is degenerate, i.e. if *min_ellipse* is empty, equal to a single point or equal to a segment, equivalently if the number of support points is less than 3.

Modifiers

New points can be added to an existing *min_ellipse*, allowing to build $me(P)$ incrementally, e.g. if P is not known in advance. Compared to the direct creation of $me(P)$, this is not much slower, because the construction method is incremental itself.

void *min_ellipse.insert(Point p)* inserts *p* into *min_ellipse* and recomputes the smallest enclosing ellipse.

template < class InputIterator >
void *min_ellipse.insert(InputIterator first, InputIterator last)*

inserts the points in the range [*first*,*last*) into *min_ellipse* and recomputes the smallest enclosing ellipse by calling *insert(p)* for each point *p* in [*first*,*last*).
Requirement: The value type of *first* and *last* is *Point*.

void *min_ellipse.clear()* deletes all points in *min_ellipse* and sets *min_ellipse* to the empty set.
Postcondition: *min_ellipse.is_empty()* = *true*.

Validity Check

An object *min_ellipse* is valid, iff

- *min_ellipse* contains all points of its defining set *P*,
- *min_ellipse* is the smallest ellipse spanned by its support set *S*, and
- *S* is minimal, i.e. no support point is redundant.

Note: In this release only the first item is considered by the validity check.

bool *min_ellipse.is_valid(bool verbose = false, int level = 0) const*

returns *true*, iff *min_ellipse* contains all points of its defining set *P*. If *verbose* is *true*, some messages concerning the performed checks are written to standard error stream. The second parameter *level* is not used, we provide it only for consistency with interfaces of other classes.

Miscellaneous

const Traits&

min_ellipse.traits() const returns a const reference to the traits class object.

I/O

std::ostream&

std::ostream& os << min_ellipse

writes *min_ellipse* to output stream *os*.
Requirement: The output operator is defined for *Point* (and for *Ellipse*, if pretty printing is used).

std::istream&

std::istream& is >> min_ellipse&

reads *min_ellipse* from input stream *is*.
Requirement: The input operator is defined for *Point*.

See Also

CGAL::Min_circle_2<Traits> . . . . page 3802
CGAL::Min_ellipse_2_traits_2<K> . . . . page 3819
MinEllipse2Traits . . . . page 3821

Implementation

We implement the incremental algorithm of Welzl, with move-to-front heuristic [Wel91], using the primitives as described in [GS97a, GS97b]. The whole implementation is described in [GS98b].

If randomization is chosen, the creation time is almost always linear in the number of points. Access functions and predicates take constant time, inserting a point might take up to linear time, but substantially less than computing the new smallest enclosing ellipse from scratch. The clear operation and the check for validity each takes linear time.

Example

To illustrate the usage of *Min_ellipse_2<Traits>* and to show that randomization can be useful in certain cases, we give an example. The example also shows how the coefficents of the constructed ellipse can be accessed.

```
#include <CGAL/Cartesian.h>
#include <CGAL/Min_ellipse_2.h>
#include <CGAL/Min_ellipse_2_traits_2.h>
#include <CGAL/Gmpq.h>

#include <cassert>

typedef  CGAL::Gmpq                       NT;
typedef  CGAL::Cartesian<NT>              K;
typedef  CGAL::Point_2<K>                 Point;
typedef  CGAL::Min_ellipse_2_traits_2<K>  Traits;
typedef  CGAL::Min_ellipse_2<Traits>      Min_ellipse;


int
main( int, char**)
{
    const int n = 200;
    Point     P[n];

    for ( int i = 0; i < n; ++i)
        P[ i] = Point( i % 2 ? i : -i , 0);
    // (0,0), (-1,0), (2,0), (-3,0)

    std::cout << "Computing ellipse (without randomization)...";
    std::cout.flush();
    Min_ellipse  me1( P, P+n, false);    // very slow
    std::cout << "done." << std::endl;

    std::cout << "Computing ellipse (with randomization)...";
    std::cout.flush();
    Min_ellipse  me2( P, P+n, true);     // fast
    std::cout << "done." << std::endl;

    // because all input points are collinear, the ellipse is
    // degenerate and equals a line segment; the ellipse has
    // two support points
    assert(me2.is_degenerate());
    assert(me2.number_of_support_points()==2);

    // prettyprinting
    CGAL::set_pretty_mode( std::cout);
    std::cout << me2;

    // in general, the ellipse is not explicitly representable
    // over the input number type NT; when you use the default
    // traits class CGAL::Min_ellipse_2_traits_2<K>, you can
    // get double approximations for the coefficients of the
    // underlying conic curve. NOTE: this curve only exists
    // in the nondegenerate case!
```

```
    me2.insert(Point(0,1)); // resolves the degeneracy
    assert(!me2.is_degenerate());

    // get the coefficients
    double r,s,t,u,v,w;
    me2.ellipse().double_coefficients( r, s, t, u, v, w);
    std::cout << "ellipse has the equation " <<
      r << " x^2 + " <<
      s << " y^2 + " <<
      t << " xy + " <<
      u << " x + " <<
      v << " y + " <<
      w << " = 0." << std::endl;

    return 0;
}
```

File: examples/Min_ellipse_2/min_ellipse_2.cpp

CGAL::Min_ellipse_2_traits_2<K>

Class

Definition

The class *Min_ellipse_2_traits_2<K>* is a traits class for *CGAL::Min_ellipse_2<Traits>* using the two-dimensional CGAL kernel.

#include <CGAL/Min_ellipse_2_traits_2.h>

Requirements

The template parameter *K* is a model for *Kernel*.

Is Model for the Concepts

MinEllipse2Traits . . . . page 3821

Types

Min_ellipse_2_traits_2<K>:: Point	typedef to *K::Point_2*.
Min_ellipse_2_traits_2<K>:: Ellipse	internal type.

The Ellipse type provides the following access methods not required by the concept *MinEllipse2Traits*.

bool	*ellipse.is_circle()*	tests whether the ellipse is a circle.
void	*ellipse.double_coefficients(double &r, double &s, double &t, double &u, double &v, double &w)*	gives a double approximation of the ellipse's conic equation. If *K* is a Cartesian kernel, the ellipse is the set of all points (x,y) satisfying $rx^2+sy^2+txy+ux+vy+w=0$. In the Homogeneous case, the ellipse is the set of points (hx,hy,hw) satisfying $r(hx)^2+s(hy)^2+t(hx)(hy)+u(hx)(hw)+v(hy)(hw)+w(hw)^2=0$.

Creation

Min_ellipse_2_traits_2<K> traits; default constructor.

Min_ellipse_2_traits_2<K> traits(Min_ellipse_2_traits_2<K>);

copy constructor.

See Also

CGAL::Min_ellipse_2<Traits> . . . . page 3812
MinEllipse2Traits . . . . page 3821

MinEllipse2Traits

Concept

Definition

This concept defines the requirements for traits classes of *CGAL::Min_ellipse_2<Traits>*.

Types

MinEllipse2Traits:: Point	The point type must provide default and copy constructor, assignment and equality test.
MinEllipse2Traits:: Ellipse	The ellipse type must fulfill the requirements listed below in the next section.

Variables

Ellipse	*ellipse;*	The current ellipse. This variable is maintained by the algorithm, the user should neither access nor modify it directly.

Creation

Only default and copy constructor are required.

MinEllipse2Traits traits;

MinEllipse2Traits traits(Traits);

Has Models

CGAL::Min_ellipse_2_traits_2<K> . . . . page 3819

See Also

CGAL::Min_ellipse_2<Traits> . . . . page 3812

Ellipse Type (*Ellipse*)

Definition

An object of the class *Ellipse* is an ellipse in two-dimensional Euclidean plane $\mathbb{E}^2$. Its boundary splits the plane into a bounded and an unbounded side. By definition, an empty *Ellipse* has no boundary and no bounded side, i.e. its unbounded side equals the whole plane $\mathbb{E}^2$.

Types

Ellipse:: Point — Point type.

Creation

void *ellipse.set()* — sets *ellipse* to the empty ellipse.

void *ellipse.set(Point p)* — sets *ellipse* to the ellipse containing exactly $\{p\}$.

void *ellipse.set(Point p, Point q)*

sets *ellipse* to the ellipse containing exactly the segment connecting p and q. The algorithm guarantees that *set* is never called with two equal points.

void *ellipse.set(Point p, Point q, Point r)*

sets *ellipse* to the smallest ellipse through p,q,r. The algorithm guarantees that *set* is never called with three collinear points.

void *ellipse.set(Point p, Point q, Point r, Point s)*

sets *ellipse* to the smallest ellipse through p,q,r,s. The algorithm guarantees that this ellipse exists.

void *ellipse.set(Point p, Point q, Point r, Point s, Point t)*

sets *ellipse* to the unique conic through p,q,r,s,t. The algorithm guarantees that this conic is an ellipse.

Predicates

bool *ellipse.has_on_unbounded_side(Point p) const*

returns *true*, iff p lies properly outside of *ellipse*.

Each of the following predicates is only needed, if the corresponding predicate of *Min_ellipse_2* is used.

CGAL::Bounded_side

ellipse.bounded_side(Point p) const

returns *CGAL::ON_BOUNDED_SIDE*, *CGAL::ON_BOUNDARY*, or *CGAL::ON_UNBOUNDED_SIDE* iff *p* lies properly inside, on the boundary, or properly outside of *ellipse*, resp.

bool *ellipse.has_on_bounded_side(Point p) const*

returns *true*, iff *p* lies properly inside *ellipse*.

bool *ellipse.has_on_boundary(Point p) const*

returns *true*, iff *p* lies on the boundary of *ellipse*.

bool *ellipse.is_empty() const*

returns *true*, iff *ellipse* is empty (this implies degeneracy).

bool *ellipse.is_degenerate() const*

returns *true*, iff *ellipse* is degenerate, i.e. if *ellipse* is empty or equal to a single point.

I/O

The following I/O operator is only needed, if the corresponding I/O operator of *Min_ellipse_2* is used.

ostream& *ostream& os << ellipse*

writes *ellipse* to output stream *os*.

Function

CGAL::min_rectangle_2

Definition

The function computes a minimum area enclosing rectangle $R(P)$ of a given convex point set P. Note that $R(P)$ is not necessarily axis-parallel, and it is in general not unique. The focus on convex sets is no restriction, since any rectangle enclosing P – as a convex set – contains the convex hull of P. For general point sets one has to compute the convex hull as a preprocessing step.

#include <CGAL/min_quadrilateral_2.h>

template < class ForwardIterator, class OutputIterator, class Traits >
OutputIterator *min_rectangle_2(*
ForwardIterator points_begin,
ForwardIterator points_end,
OutputIterator o,
Traits& t = Default_traits)

computes a minimum area enclosing rectangle of the point set described by [*points_begin*, *points_end*), writes its vertices (counterclockwise) to *o*, and returns the past-the-end iterator of this sequence.
If the input range is empty, *o* remains unchanged.
If the input range consists of one element only, this point is written to *o* four times.

Precondition: The points denoted by the range [*points_begin*, *points_end*) form the boundary of a simple convex polygon P in counterclockwise orientation.

The geometric types and operations to be used for the computation are specified by the traits class parameter *t*. The parameter can be omitted, if *ForwardIterator* refers to a two-dimensional point type from one the CGAL kernels. In this case, a default traits class (*Min_quadrilateral_default_traits_2<K>*) is used.

Requirement:

1. If *Traits* is specified, it is a model for *MinQuadrilateralTraits_2* and the value type *VT* of *ForwardIterator* is *Traits::Point_2*. Otherwise *VT* is *CGAL::Point_2<K>* for some kernel *K*.
2. *OutputIterator* accepts *VT* as value type.

See Also

CGAL::min_parallelogram_2 . . . page 3826
CGAL::min_strip_2 . . . page 3828
MinQuadrilateralTraits_2 . . . page 3834
CGAL::Min_quadrilateral_default_traits_2<K> . . . page 3830

Implementation

We use a rotating caliper algorithm [Tou83] with worst case running time linear in the number of input points.

Example

The following code generates a random convex polygon P with 20 vertices and computes the minimum enclosing rectangle of P.

```
#include <CGAL/Cartesian.h>
#include <CGAL/Polygon_2.h>
#include <CGAL/point_generators_2.h>
#include <CGAL/random_convex_set_2.h>
#include <CGAL/min_quadrilateral_2.h>
#include <iostream>

struct Kernel : public CGAL::Cartesian<double> {};

typedef Kernel::Point_2                           Point_2;
typedef Kernel::Line_2                            Line_2;
typedef CGAL::Polygon_2<Kernel>                   Polygon_2;
typedef CGAL::Random_points_in_square_2<Point_2>  Generator;

int main()
{
  // build a random convex 20-gon p
  Polygon_2 p;
  CGAL::random_convex_set_2(20, std::back_inserter(p), Generator(1.0));
  std::cout << p << std::endl;

  // compute the minimal enclosing rectangle p_m of p
  Polygon_2 p_m;
  CGAL::min_rectangle_2(
    p.vertices_begin(), p.vertices_end(), std::back_inserter(p_m));
  std::cout << p_m << std::endl;

  return 0;
}
```

File: examples/Min_quadrilateral_2/minimum_enclosing_rectangle_2.cpp

Function

CGAL::min_parallelogram_2

Definition

The function computes a minimum area enclosing parallelogram $A(P)$ of a given convex point set P. Note that $R(P)$ is not necessarily axis-parallel, and it is in general not unique. The focus on convex sets is no restriction, since any parallelogram enclosing P – as a convex set – contains the convex hull of P. For general point sets one has to compute the convex hull as a preprocessing step.

#include <CGAL/min_quadrilateral_2.h>

```
template < class ForwardIterator, class OutputIterator, class Traits >
OutputIterator        min_parallelogram_2(
                              ForwardIterator points_begin,
                              ForwardIterator points_end,
                              OutputIterator o,
                              Traits& t = Default_traits)
```

computes a minimum area enclosing parallelogram of the point set described by [*points_begin*, *points_end*), writes its vertices (counterclockwise) to *o* and returns the past-the-end iterator of this sequence. If the input range is empty, *o* remains unchanged.
If the input range consists of one element only, this point is written to *o* four times.

Precondition: The points denoted by the range [*points_begin*, *points_end*) form the boundary of a simple convex polygon P in counterclockwise orientation.

The geometric types and operations to be used for the computation are specified by the traits class parameter *t*. The parameter can be omitted, if *ForwardIterator* refers to a two-dimensional point type from one the CGAL kernels. In this case, a default traits class (*Min_quadrilateral_default_traits_2<K>*) is used.

Requirement:

1. If *Traits* is specified, it is a model for *MinQuadrilateralTraits_2* and the value type *VT* of *ForwardIterator* is *Traits::Point_2*. Otherwise *VT* is *CGAL::Point_2<K>* for some kernel *K*.
2. *OutputIterator* accepts *VT* as value type.

See Also

CGAL::min_rectangle_2 .... page 3824
CGAL::min_strip_2 .... page 3828
MinQuadrilateralTraits_2 .... page 3834
CGAL::Min_quadrilateral_default_traits_2<K> .... page 3830

Implementation

We use a rotating caliper algorithm [STV+95, Vai90] with worst case running time linear in the number of input points.

Example

The following code generates a random convex polygon P with 20 vertices and computes the minimum enclosing parallelogram of P.

```
#include <CGAL/Cartesian.h>
#include <CGAL/Polygon_2.h>
#include <CGAL/point_generators_2.h>
#include <CGAL/random_convex_set_2.h>
#include <CGAL/min_quadrilateral_2.h>
#include <iostream>

struct Kernel : public CGAL::Cartesian<double> {};

typedef Kernel::Point_2                          Point_2;
typedef Kernel::Line_2                           Line_2;
typedef CGAL::Polygon_2<Kernel>                  Polygon_2;
typedef CGAL::Random_points_in_square_2<Point_2> Generator;

int main()
{
  // build a random convex 20-gon p
  Polygon_2 p;
  CGAL::random_convex_set_2(20, std::back_inserter(p), Generator(1.0));
  std::cout << p << std::endl;

  // compute the minimal enclosing parallelogram p_m of p
  Polygon_2 p_m;
  CGAL::min_parallelogram_2(
    p.vertices_begin(), p.vertices_end(), std::back_inserter(p_m));
  std::cout << p_m << std::endl;

  return 0;
}
```

File: examples/Min_quadrilateral_2/minimum_enclosing_parallelogram_2.cpp

Function

CGAL::min_strip_2

Definition

The function computes a minimum width enclosing strip $S(P)$ of a given convex point set P. A strip is the closed region bounded by two parallel lines in the plane. Note that $S(P)$ is not unique in general. The focus on convex sets is no restriction, since any parallelogram enclosing P – as a convex set – contains the convex hull of P. For general point sets one has to compute the convex hull as a preprocessing step.

#include <CGAL/min_quadrilateral_2.h>

template < class ForwardIterator, class OutputIterator, class Traits >
OutputIterator *min_strip_2(*
ForwardIterator points_begin,
ForwardIterator points_end,
OutputIterator o,
Traits& t = Default_traits)

computes a minimum enclosing strip of the point set described by [*points_begin*, *points_end*), writes its two bounding lines to *o* and returns the past-the-end iterator of this sequence.
If the input range is empty or consists of one element only, *o* remains unchanged.

Precondition: The points denoted by the range [*points_begin*, *points_end*) form the boundary of a simple convex polygon P in counterclockwise orientation.

The geometric types and operations to be used for the computation are specified by the traits class parameter *t*. The parameter can be omitted, if *ForwardIterator* refers to a two-dimensional point type from one the CGAL kernels. In this case, a default traits class (*Min_quadrilateral_default_traits_2<K>*) is used.

Requirement:

1. If *Traits* is specified, it is a model for *MinQuadrilateralTraits_2* and the value type *VT* of *ForwardIterator* is *Traits::Point_2*. Otherwise *VT* is *CGAL::Point_2<K>* for some kernel *K*.
2. *OutputIterator* accepts *Traits::Line_2* as value type.

See Also

CGAL::min_rectangle_2 . . . . . . page 3824
CGAL::min_parallelogram_2 . . . . . . page 3826
MinQuadrilateralTraits_2 . . . . . . page 3834
CGAL::Min_quadrilateral_default_traits_2<K> . . . . . . page 3830

Implementation

We use a rotating caliper algorithm [Tou83] with worst case running time linear in the number of input points.

Example

The following code generates a random convex polygon P with 20 vertices and computes the minimum enclosing strip of P.

```
#include <CGAL/Cartesian.h>
#include <CGAL/Polygon_2.h>
#include <CGAL/point_generators_2.h>
#include <CGAL/random_convex_set_2.h>
#include <CGAL/min_quadrilateral_2.h>
#include <iostream>

struct Kernel : public CGAL::Cartesian<double> {};

typedef Kernel::Point_2                              Point_2;
typedef Kernel::Line_2                               Line_2;
typedef CGAL::Polygon_2<Kernel>                      Polygon_2;
typedef CGAL::Random_points_in_square_2<Point_2>    Generator;

int main()
{
  // build a random convex 20-gon p
  Polygon_2 p;
  CGAL::random_convex_set_2(20, std::back_inserter(p), Generator(1.0));
  std::cout << p << std::endl;

  // compute the minimal enclosing strip p_m of p
  Line_2 p_m[2];
  CGAL::min_strip_2(p.vertices_begin(), p.vertices_end(), p_m);
  std::cout << p_m[0] << "\n" << p_m[1] << std::endl;

  return 0;
}
```

File: examples/Min_quadrilateral_2/minimum_enclosing_strip_2.cpp

CGAL::Min_quadrilateral_default_traits_2<K>

Definition

The class *Min_quadrilateral_default_traits_2<K>* is a traits class for the functions *min_rectangle_2*, *min_parallelogram_2* and *min_strip_2* using a two-dimensional CGAL kernel.

Requirements

The template parameter *K* is a model for *Kernel*.

#include <CGAL/Min_quadrilateral_traits_2.h>

Is Model for the Concepts

MinQuadrilateralTraits_2 . . . . page 3834

Types

Min_quadrilateral_default_traits_2<K>:: Point_2	Kernel::Point_2 . . . . page 794.
Min_quadrilateral_default_traits_2<K>:: Vector_2	Kernel::Vector_2 . . . . page 806.
Min_quadrilateral_default_traits_2<K>:: Direction_2	Kernel::Direction_2 . . . . page 785.
Min_quadrilateral_default_traits_2<K>:: Line_2	Kernel::Line_2 . . . . page 789.
Min_quadrilateral_default_traits_2<K>:: Rectangle_2	internal type.
Min_quadrilateral_default_traits_2<K>:: Parallelogram_2	internal type.
Min_quadrilateral_default_traits_2<K>:: Strip_2	internal type.

Predicates

Min_quadrilateral_default_traits_2<K>:: Equal_2	Kernel::Equal_2 . . . . page 996.

Min_quadrilateral_default_traits_2<K>:: Less_xy_2 Kernel::Less_xy_2 . . . . page **??**.

Min_quadrilateral_default_traits_2<K>:: Less_yx_2 Kernel::Less_yx_2 . . . . page **??**.

Min_quadrilateral_default_traits_2<K>:: Orientation_2

Kernel::Orientation_2 . . . . page 1034.

Min_quadrilateral_default_traits_2<K>:: Has_on_negative_side_2

Kernel::Has_on_negative_side_2 . . . . page **??**.

Min_quadrilateral_default_traits_2<K>:: Compare_angle_with_x_axis_2

Kernel::Compare_angle_with_x_axis_2 . . . . page **??**.

Min_quadrilateral_default_traits_2<K>:: Area_less_rectangle_2

AdaptableBinaryFunction class
op: *Rectangle_2* $\times$ *Rectangle_2* $\rightarrow$ *bool*.
op(r1,r2) returns true, iff the area of $r1$ is strictly less than the area of $r2$.

Min_quadrilateral_default_traits_2<K>:: Area_less_parallelogram_2

AdaptableBinaryFunction class
op: *Parallelogram_2* $\times$ *Parallelogram_2* $\rightarrow$ *bool*.
op(p1,p2) returns true, iff the area of $p1$ is strictly less than the area of $p2$.

Min_quadrilateral_default_traits_2<K>:: Width_less_strip_2

AdaptableBinaryFunction class
op: *Strip_2* $\times$ *Strip_2* $\rightarrow$ *bool*.
op(s1,s2) returns true, iff the width of $s1$ is strictly less than the width of $s2$.

Constructions

Min_quadrilateral_default_traits_2<K>:: Construct_vector_2

Kernel::Construct_vector_2 . . . . page **??**.

Min_quadrilateral_default_traits_2<K>:: Construct_vector_from_direction_2

AdaptableFunctor
op: *Direction_2* $\rightarrow$ *Vector_2*.
op(d) returns a vector in direction d.

Min_quadrilateral_default_traits_2<K>:: Construct_perpendicular_vector_2

Kernel::Construct_perpendicular_vector_2 . . . . page **??**.

Min_quadrilateral_default_traits_2<K>:: Construct_direction_2

Kernel::Construct_direction_2 . . . . page **??**.

Min_quadrilateral_default_traits_2<K>:: Construct_opposite_direction_2

Kernel::Construct_opposite_direction_2 . . . . . . page **??**.

Min_quadrilateral_default_traits_2<K>:: Construct_line_2

Kernel::Construct_line_2 . . . . page **??**.

Min_quadrilateral_default_traits_2<K>:: Construct_rectangle_2

Function class
op: $Point\_2 \times Direction\_2 \times Point\_2 \times Point\_2 \times Point\_2 \rightarrow Rectangle\_2$.
If the points $p1, p2, p3, p4$ form the boundary of a convex polygon (oriented counterclockwise), *op(p1,d,p2,p3,p4)* returns the rectangle with one of the points on each side and one sides parallel to d.

Min_quadrilateral_default_traits_2<K>:: Construct_parallelogram_2

Function class
op: $Point\_2 \times Direction\_2 \times Point\_2 \times Direction\_2 \times Point\_2 \times Point\_2 \rightarrow Rectangle\_2$.
If the points $p1, p2, p3, p4$ form the boundary of a convex polygon (oriented counterclockwise), *op(p1,d1,p2,d2,p3,p4)* returns the parallelogram with one of the points on each side and one side parallel to each of $d1$ and $d2$.

Min_quadrilateral_default_traits_2<K>:: Construct_strip_2

Function class
op: $Point\_2 \times Direction\_2 \times Point\_2 \rightarrow Strip\_2$.
op(p1,d,p2) returns the strip bounded by the lines through $p1$ resp. $p2$ with direction d.

Operations

template < class OutputIterator >

OutputIterator *t.copy_rectangle_vertices_2(const Rectangle_2& r, OutputIterator o) const*

copies the four vertices of *r* in counterclockwise order to *o*.

template < class OutputIterator >
OutputIterator *t.copy_parallelogram_vertices_2(const Parallelogram_2& p, OutputIterator o) const*

copies the four vertices of *p* in counterclockwise order to *o*.

template < class OutputIterator >
OutputIterator *t.copy_strip_lines_2(const Strip_2& s, OutputIterator o) const*

copies the two lines bounding *s* to *o*.

Additionally, for each of the predicate and construction functor types listed above, there is a member function that requires no arguments and returns an instance of that functor type. The name of the member function is the uncapitalized name of the type returned with the suffix *_object* appended. For example, for the functor type *Construct_vector_2* the following member function exists:

Construct_vector_2

t.construct_vector_2_object() const

See Also

CGAL::min_rectangle_2 .... page 3824
CGAL::min_parallelogram_2 .... page 3826
CGAL::min_strip_2 .... page 3828

MinQuadrilateralTraits_2

Definition

The concept MinQuadrilateralTraits_2 defines types and operations needed to compute minimum enclosing quadrilaterals of a planar point set using the functions *min_rectangle_2*, *min_parallelogram_2* and *min_strip_2*.

Types

MinQuadrilateralTraits_2:: Point_2	type for representing points.
MinQuadrilateralTraits_2:: Vector_2	type for representing vectors.
MinQuadrilateralTraits_2:: Direction_2	type for representing directions.
MinQuadrilateralTraits_2:: Line_2	type for representing lines.
MinQuadrilateralTraits_2:: Rectangle_2	type for representing (not necessarily axis-parallel) rectangles.
MinQuadrilateralTraits_2:: Parallelogram_2	type for representing parallelograms.
MinQuadrilateralTraits_2:: Strip_2	type for representing strips, that is the closed region bounded by two parallel lines.

Predicates

MinQuadrilateralTraits_2:: Equal_2 — a model for Kernel::Equal_2 . . . . page 996.

MinQuadrilateralTraits_2:: Less_xy_2 — a model for Kernel::Less_xy_2 . . . . page **??**.

MinQuadrilateralTraits_2:: Less_yx_2 — a model for Kernel::Less_yx_2 . . . . page **??**.

MinQuadrilateralTraits_2:: Has_on_negative_side_2

a model for Kernel::Has_on_negative_side_2 . . . . . page **??**.

MinQuadrilateralTraits_2:: Compare_angle_with_x_axis_2

a model for Kernel::Compare_angle_with_x_axis_2page **??**.

MinQuadrilateralTraits_2:: Area_less_rectangle_2

AdaptableFunctor
op: $Rectangle\_2 \times Rectangle\_2 \rightarrow bool$.
op(r1,r2) returns true, iff the area of $r1$ is strictly less than the area of $r2$.

MinQuadrilateralTraits_2:: Area_less_parallelogram_2

AdaptableFunctor
op: *Parallelogram_2* × *Parallelogram_2* → *bool*.
op(p1,p2) returns true, iff the area of $p1$ is strictly less than the area of $p2$.

MinQuadrilateralTraits_2:: Width_less_strip_2

AdaptableFunctor
op: *Strip_2* × *Strip_2* → *bool*.
op(s1,s2) returns true, iff the width of $s1$ is strictly less than the width of $s2$.

The following type is used for expensive precondition checking only.

MinQuadrilateralTraits_2:: Orientation_2

a model for Kernel::Orientation_2 . . . . page 1034.

Constructions

MinQuadrilateralTraits_2:: Construct_vector_2

a model for Kernel::Construct_vector_2 . . . . page **??**.

MinQuadrilateralTraits_2:: Construct_vector_from_direction_2

AdaptableFunctor
op: *Direction_2* → *Vector_2*.
op(d) returns a vector in direction d.

MinQuadrilateralTraits_2:: Construct_perpendicular_vector_2

a model for Kernel::Construct_perpendicular_vector_2 page **??**.

MinQuadrilateralTraits_2:: Construct_direction_2

a model for Kernel::Construct_direction_2 . . . . page **??**.

MinQuadrilateralTraits_2:: Construct_opposite_direction_2

a model for Kernel::Construct_opposite_direction_2 page **??**.

MinQuadrilateralTraits_2:: Construct_line_2

a model for Kernel::Construct_line_2 . . . . page **??**.

MinQuadrilateralTraits_2:: Construct_rectangle_2

Function class
op: $Point\_2 \times Direction\_2 \times Point\_2 \times Point\_2 \times Point\_2 \rightarrow Rectangle\_2$.
If the points $p1, p2, p3, p4$ form the boundary of a convex polygon (oriented counterclockwise), *op(p1,d,p2,p3,p4)* returns the rectangle with one of the points on each side and one sides parallel to d.

MinQuadrilateralTraits_2:: Construct_parallelogram_2

Function class
op: $Point\_2 \times Direction\_2 \times Point\_2 \times Direction\_2 \times Point\_2 \times Point\_2 \rightarrow Rectangle\_2$.
If the points $p1, p2, p3, p4$ form the boundary of a convex polygon (oriented counterclockwise), *op(p1,d1,p2,d2,p3,p4)* returns the parallelogram with one of the points on each side and one side parallel to each of $d1$ and $d2$.

MinQuadrilateralTraits_2:: Construct_strip_2

Function class
op: $Point\_2 \times Direction\_2 \times Point\_2 \rightarrow Strip\_2$.
op(p1,d,p2) returns the strip bounded by the lines through $p1$ resp. $p2$ with direction d.

Operations

template < class OutputIterator >
OutputIterator *t.copy_rectangle_vertices_2(const Rectangle_2& r, OutputIterator o) const*

copies the four vertices of r in counterclockwise order to o.

template < class OutputIterator >
OutputIterator *t.copy_parallelogram_vertices_2(const Parallelogram_2& p, OutputIterator o) const*

copies the four vertices of p in counterclockwise order to o.

template < class OutputIterator >
OutputIterator *t.copy_strip_lines_2(const Strip_2& s, OutputIterator o) const*

copies the two lines bounding s to o.

Additionally, for each of the predicate and construction functor types listed above, there must exist a member function that requires no arguments and returns an instance of that functor type. The name of the member function is the uncapitalized name of the type returned with the suffix *_object* appended. For example, for the functor type *Construct_vector_2* the following member function must exist:

Construct_vector_2

t.construct_vector_2_object() const

Has Models

CGAL::Min_quadrilateral_default_traits_2<K> . . . . page 3830

See Also

CGAL::min_rectangle_2 . . . . page 3824
CGAL::min_parallelogram_2 . . . . page 3826
CGAL::min_strip_2 . . . . page 3828

Function

CGAL::rectangular_p_center_2

Definition

The function *rectangular_p_center_2* computes rectilinear p-centers of a planar point set, i.e. a set of p points such that the maximum minimal L_∞-distance between both sets is minimized.

More formally the problem can be defined as follows.

Given a finite set $\mathcal{P}$ of points, compute a point set $\mathcal{C}$ with $|\mathcal{C}| \leq p$ such that the p-radius of $\mathcal{P}$,

$$rad_p(\mathcal{P}) := \max_{P \in \mathcal{P}} \min_{Q \in \mathcal{C}} ||P - Q||_\infty$$

is minimized. We can interpret $\mathcal{C}$ as the best approximation (with respect to the given metric) for $\mathcal{P}$ with at most p points.

#include <CGAL/rectangular_p_center_2.h>

```
template < class ForwardIterator, class OutputIterator, class FT, class Traits >
OutputIterator        rectangular_p_center_2(
                              ForwardIterator f,
                              ForwardIterator l,
                              OutputIterator o,
                              FT& r,
                              int p,
                              Traits t = Default_traits)
```

computes rectilinear p-centers for the point set described by the range $[f, l)$, sets r to the corresponding p-radius, writes the at most p center points to o and returns the past-the-end iterator of this sequence.

Precondition: $2 \leq p \leq 4$.

The geometric types and operations to be used for the computation are specified by the traits class parameter t. This parameter can be omitted if *ForwardIterator* refers to a point type from the 2D-Kernel. In this case, a default traits class (*Rectangular_p_center_default_traits_2<K>*) is used.

Requirement:

1. *Either: (if no traits parameter is given)* Value type of *ForwardIterator* is *CGAL::Point_2<K>* for some representation class *K* and *FT* is equivalent to *K::FT*,
2. *Or: (if a traits parameter is specified) Traits* is a model for *RectangularPCenterTraits_2*.
3. *OutputIterator* accepts the value type of *ForwardIterator* as value type.

See Also

RectangularPCenterTraits_2 . . . . page 3843
CGAL::Rectangular_p_center_default_traits_2<K> . . . . page 3840
CGAL::sorted_matrix_search . . . . page 465

Implementation

The runtime is linear for $p \in \{2,3\}$ and $O(n \cdot \log n)$ for $p = 4$ where n is the number of input points. These runtimes are worst case optimal. The 3-center algorithm uses a prune-and-search technique described in [Hof99]. The 4-center implementation uses sorted matrix search [FJ83, FJ84] and fast algorithms for piercing rectangles [SW96].

Example

The following code generates a random set of ten points and computes its two-centers.

```
#include <CGAL/Cartesian.h>
#include <CGAL/point_generators_2.h>
#include <CGAL/rectangular_p_center_2.h>
#include <CGAL/IO/Ostream_iterator.h>
#include <CGAL/algorithm.h>
#include <iostream>
#include <algorithm>
#include <vector>

typedef double                                          FT;

typedef CGAL::Cartesian<FT>                             Kernel;

typedef Kernel::Point_2                                 Point;
typedef std::vector<Point>                              Cont;
typedef CGAL::Random_points_in_square_2<Point>          Generator;
typedef CGAL::Ostream_iterator<Point,std::ostream>      OIterator;

int main()
{
  int n = 10;
  int p = 2;
  OIterator cout_ip(std::cout);
  CGAL::set_pretty_mode(std::cout);

  Cont points;
  CGAL::cpp0x::copy_n(Generator(1), n, std::back_inserter(points));
  std::cout << "Generated Point Set:\n";
  std::copy(points.begin(), points.end(), cout_ip);

  FT p_radius;
  std::cout << "\n\n" << p << "-centers:\n";
  CGAL::rectangular_p_center_2(
    points.begin(), points.end(), cout_ip, p_radius, 3);
  std::cout << "\n\n" << p << "-radius = " << p_radius << std::endl;

  return 0;
}
```

File: examples/Matrix_search/rectangular_p_center_2.cpp

Class

CGAL::Rectangular_p_center_default_traits_2<K>

Definition

The class *Rectangular_p_center_default_traits_2<K>* defines types and operations needed to compute rectilinear p-centers of a planar point set using the function *rectangular_p_center_2*.

Requirements

The template parameter K is a model for *Kernel*.

Is Model for the Concepts

RectangularPCenterTraits_2 . . . . page 3843

Types

Rectangular_p_center_default_traits_2<K>:: FT

typedef to *K::FT*.

Rectangular_p_center_default_traits_2<K>:: Point_2

typedef to *K::Point_2*.

Rectangular_p_center_default_traits_2<K>:: Iso_rectangle_2

typedef to *K::Iso_rectangle_2*.

Rectangular_p_center_default_traits_2<K>:: Less_x_2

typedef to *K::Less_x_2*.

Rectangular_p_center_default_traits_2<K>:: Less_y_2

typedef to *K::Less_y_2*.

Rectangular_p_center_default_traits_2<K>:: Construct_vertex_2

typedef to *K::Construct_vertex_2*.

Rectangular_p_center_default_traits_2<K>:: Construct_iso_rectangle_2

typedef to *K::Construct_iso_rectangle_2*.

Rectangular_p_center_default_traits_2<K>:: Signed_x_distance_2

adaptable binary function class: $Point\_2 \times Point\_2 \rightarrow FT$ returns the signed distance of two points' x-coordinates.

Rectangular_p_center_default_traits_2<K>:: Signed_y_distance_2

adaptable binary function class: $Point\_2 \times Point\_2 \rightarrow FT$ returns the signed distance of two points' y-coordinates.

Rectangular_p_center_default_traits_2<K>:: Infinity_distance_2

adaptable binary function class: $Point\_2 \times Point\_2 \rightarrow FT$ returns the $||\cdot||_\infty$ distance of two points.

Rectangular_p_center_default_traits_2<K>:: Signed_infinity_distance_2

adaptable binary function class: $Point\_2 \times Point\_2 \rightarrow FT$ returns the signed $||\cdot||_\infty$ distance of two points.

Rectangular_p_center_default_traits_2<K>:: Construct_point_2_below_left_implicit_point_2

3-argument function class: $Point\_2 \times Point\_2 \times FT \rightarrow Point\_2$. For arguments (p,q,r) it returns the lower-left corner of the iso-oriented square with sidelength r and upper-right corner at the intersection of the vertical line through p and the horizontal line through q.

Rectangular_p_center_default_traits_2<K>:: Construct_point_2_below_right_implicit_point_2

3-argument function class: $Point\_2 \times Point\_2 \times FT \rightarrow Point\_2$. For arguments (p,q,r) it returns the lower-right corner of the iso-oriented square with sidelength r and upper-left corner at the intersection of the vertical line through p and the horizontal line through q.

Rectangular_p_center_default_traits_2<K>:: Construct_point_2_above_right_implicit_point_2

3-argument function class: $Point\_2 \times Point\_2 \times FT \rightarrow Point\_2$. For arguments (p,q,r) it returns the upper-right corner of the iso-oriented square with sidelength r and lower-left corner at the intersection of the vertical line through p and the horizontal line through q.

Rectangular_p_center_default_traits_2<K>:: Construct_point_2_above_left_implicit_point_2

3-argument function class: $Point\_2 \times Point\_2 \times FT \rightarrow Point\_2$. For arguments (p,q,r) it returns the upper-left corner of the iso-oriented square with sidelength r and lower-right corner at the intersection of the vertical line through p and the horizontal line through q.

Operations

For every function class listed above there is a member function to fetch the corresponding function object.

Inf_distance_2 *t.inf_distance_2_object() const*

Signed_inf_distance_2

t.signed_inf_distance_2_object() const

Construct_vertex_2 *t.construct_vertex_2_object() const*

Construct_iso_rectangle_2

t.construct_iso_rectangle_2_object() const

Construct_iso_rectangle_2_below_left_point_2

t.construct_iso_rectangle_2_below_left_point_2_object() const

Construct_iso_rectangle_2_above_left_point_2

t.construct_iso_rectangle_2_above_left_point_2_object() const

Construct_iso_rectangle_2_below_right_point_2

t.construct_iso_rectangle_2_below_right_point_2_object() const

Construct_iso_rectangle_2_above_right_point_2

t.construct_iso_rectangle_2_above_right_point_2_object() const

See Also

CGAL::rectangular_p_center_2 . . . . page 3838

RectangularPCenterTraits_2

Definition

The concept RectangularPCenterTraits_2 defines types and operations needed to compute rectilinear p-centers of a planar point set using the function *rectangular_p_center_2*.

Types

RectangularPCenterTraits_2:: FT — model for FieldNumberType . . . . page 149.

RectangularPCenterTraits_2:: Point_2 — model for Kernel::Point_2 . . . . page 794.

RectangularPCenterTraits_2:: Iso_rectangle_2

model for Kernel::Iso_rectangle_2 . . . . page ??.

RectangularPCenterTraits_2:: Less_x_2 — model for Kernel::Less_x_2 . . . . page ??.

RectangularPCenterTraits_2:: Less_y_2 — model for Kernel::Less_y_2 . . . . page ??.

RectangularPCenterTraits_2:: Construct_vertex_2

model for Kernel::Construct_vertex_2 . . . . page ??.

RectangularPCenterTraits_2:: Construct_iso_rectangle_2

model for Kernel::Construct_iso_rectangle_2 . . . . . . page ??.

RectangularPCenterTraits_2:: Signed_x_distance_2

adaptable binary function class: *Point_2* $\times$ *Point_2* $\rightarrow$ *FT* returns the signed distance of two points' x-coordinates.

RectangularPCenterTraits_2:: Signed_y_distance_2

adaptable binary function class: *Point_2* $\times$ *Point_2* $\rightarrow$ *FT* returns the signed distance of two points' y-coordinates.

RectangularPCenterTraits_2:: Infinity_distance_2

adaptable binary function class: *Point_2* $\times$ *Point_2* $\rightarrow$ *FT* returns the $||\cdot||_\infty$ distance of two points.

RectangularPCenterTraits_2:: Signed_infinity_distance_2

adaptable binary function class: *Point_2* $\times$ *Point_2* $\rightarrow$ *FT* returns the signed $||\cdot||_\infty$ distance of two points.

RectangularPCenterTraits_2:: Construct_point_2_below_left_implicit_point_2

3-argument function class: *Point_2* $\times$ *Point_2* $\times$ *FT* $\rightarrow$ *Point_2*. For arguments (p,q,r) it returns the lower-left corner of the iso-oriented square with sidelength r and upper-right corner at the intersection of the vertical line through p and the horizontal line through q.

RectangularPCenterTraits_2:: Construct_point_2_below_right_implicit_point_2

3-argument function class: *Point_2* $\times$ *Point_2* $\times$ *FT* $\rightarrow$ *Point_2*. For arguments (p,q,r) it returns the lower-right corner of the iso-oriented square with sidelength r and upper-left corner at the intersection of the vertical line through p and the horizontal line through q.

RectangularPCenterTraits_2:: Construct_point_2_above_right_implicit_point_2

3-argument function class: *Point_2* $\times$ *Point_2* $\times$ *FT* $\rightarrow$ *Point_2*. For arguments (p,q,r) it returns the upper-right corner of the iso-oriented square with sidelength r and lower-left corner at the intersection of the vertical line through p and the horizontal line through q.

RectangularPCenterTraits_2:: Construct_point_2_above_left_implicit_point_2

3-argument function class: *Point_2* $\times$ *Point_2* $\times$ *FT* $\rightarrow$ *Point_2*. For arguments (p,q,r) it returns the upper-left corner of the iso-oriented square with sidelength r and lower-right corner at the intersection of the vertical line through p and the horizontal line through q.

Operations

For every function class listed above there is a member function to fetch the corresponding function object.

Inf_distance_2 *t.inf_distance_2_object() const*

Signed_inf_distance_2

t.signed_inf_distance_2_object() const

Construct_vertex_2 *t.construct_vertex_2_object() const*

Construct_iso_rectangle_2

t.construct_iso_rectangle_2_object() const

Construct_iso_rectangle_2_below_left_point_2

t.construct_iso_rectangle_2_below_left_point_2_object() const

Construct_iso_rectangle_2_above_left_point_2

t.construct_iso_rectangle_2_above_left_point_2_object() const

Construct_iso_rectangle_2_below_right_point_2

t.construct_iso_rectangle_2_below_right_point_2_object() const

Construct_iso_rectangle_2_above_right_point_2

t.construct_iso_rectangle_2_above_right_point_2_object() const

Has Models

CGAL::Rectangular_p_center_default_traits_2<K> . . . . page 3840

See Also

CGAL::rectangular_p_center_2 . . . . page 3838

Class

CGAL::Min_sphere_d<Traits>

Definition

An object of the class *Min_sphere_d<Traits>* is the unique sphere of smallest volume enclosing a finite (multi)set of points in d-dimensional Euclidean space $\mathbb{E}^d$. For a set P we denote by $ms(P)$ the smallest sphere that contains all points of P. $ms(P)$ can be degenerate, i.e. $ms(P) = \emptyset$ if $P = \emptyset$ and $ms(P) = \{p\}$ if $P = \{p\}$.

An inclusion-minimal subset S of P with $ms(S) = ms(P)$ is called a *support set*, the points in S are the *support points*. A support set has size at most $d+1$, and all its points lie on the boundary of $ms(P)$. In general, neither the support set nor its size are unique.

The algorithm computes a support set S which remains fixed until the next insert or clear operation.

Please note: This class is (almost) obsolete. The class *CGAL::Min_sphere_of_spheres_d<Traits>* solves a more general problem and is faster then *Min_sphere_d<Traits>* even if used only for points as input. Most importantly, *CGAL::Min_sphere_of_spheres_d<Traits>* has a specialized implementation for floating-point arithmetic which ensures correct results in a large number of cases (including highly degenerate ones). In contrast, *Min_sphere_d<Traits>* is not reliable under floating-point computations. The only advantage of *Min_sphere_d<Traits>* over *CGAL::Min_sphere_of_spheres_d<Traits>* is that the former can deal with points in homogeneous coordinates, in which case the algorithm is division-free. Thus, *Min_sphere_d<Traits>* might still be an option in case your input number type cannot (efficiently) divide.

#include <CGAL/Min_sphere_d.h>

Requirements

The class *Min_sphere_d<Traits>* expects a model of the concept *OptimisationDTraits* as its template argument. We provide the models *CGAL::Optimisation_d_traits_2*, *CGAL::Optimisation_d_traits_3* and *CGAL::Optimisation_d_traits_d* for two-, three-, and d-dimensional points respectively.

Types

Min_sphere_d<Traits>:: Traits

Min_sphere_d<Traits>:: FT — typedef to *Traits::FT*.

Min_sphere_d<Traits>:: Point — typedef to *Traits::Point*.

Min_sphere_d<Traits>:: Point_iterator — non-mutable model of the STL concept *BidirectionalIterator* with value type *Point*. Used to access the points used to build the smallest enclosing sphere.

Min_sphere_d<Traits>:: Support_point_iterator — non-mutable model of the STL concept *BidirectionalIterator* with value type *Point*. Used to access the support points defining the smallest enclosing sphere.

Creation

Min_sphere_d<Traits> min_sphere(Traits traits = Traits());

creates a variable of type *Min_sphere_d<Traits>* and initializes it to $ms(\emptyset)$. If the traits parameter is not supplied, the class *Traits* must provide a default constructor.

template < class InputIterator >
Min_sphere_d<Traits> min_sphere(InputIterator first, InputIterator last, Traits traits = Traits());

creates a variable *min_sphere* of type *Min_sphere_d<Traits>*. It is initialized to $ms(P)$ with P being the set of points in the range [*first*,*last*).
Requirement: The value type of *first* and *last* is *Point*. If the traits parameter is not supplied, the class *Traits* must provide a default constructor.
Precondition: All points have the same dimension.

int *min_sphere.number_of_points() const*

returns the number of points of *min_sphere*, i.e. $|P|$.

int *min_sphere.number_of_support_points() const*

returns the number of support points of *min_sphere*, i.e. $|S|$.

Point_iterator *min_sphere.points_begin() const*
returns an iterator referring to the first point of *min_sphere*.

Point_iterator *min_sphere.points_end() const*
returns the corresponding past-the-end iterator.

Support_point_iterator *min_sphere.support_points_begin() const*
returns an iterator referring to the first support point of *min_sphere*.

Support_point_iterator *min_sphere.support_points_end() const*
returns the corresponding past-the-end iterator.

int *min_sphere.ambient_dimension() const*

returns the dimension of the points in P. If *min_sphere* is empty, the ambient dimension is -1.

Point *min_sphere.center() const*

returns the center of *min_sphere*.
Precondition: *min_sphere* is not empty.

FT *min_sphere.squared_radius() const*

returns the squared radius of *min_sphere*.
Precondition: *min_sphere* is not empty.

Predicates

By definition, an empty *Min_sphere_d<Traits>* has no boundary and no bounded side, i.e. its unbounded side equals the whole space $\mathbb{E}^d$.

Bounded_side *min_sphere.bounded_side(Point p) const*

returns *CGAL::ON_BOUNDED_SIDE*, *CGAL::ON_BOUNDARY*, or *CGAL::ON_UNBOUNDED_SIDE* iff *p* lies properly inside, on the boundary, or properly outside of *min_sphere*, resp.
Precondition: if *min_sphere* is not empty, the dimension of *p* equals *ambient_dimension()*.

bool *min_sphere.has_on_bounded_side(Point p) const*

returns *true*, iff *p* lies properly inside *min_sphere*.
Precondition: if *min_sphere* is not empty, the dimension of *p* equals *ambient_dimension()*.

bool *min_sphere.has_on_boundary(Point p) const*

returns *true*, iff *p* lies on the boundary of *min_sphere*.
Precondition: if *min_sphere* is not empty, the dimension of *p* equals *ambient_dimension()*.

bool *min_sphere.has_on_unbounded_side(Point p) const*

returns *true*, iff *p* lies properly outside of *min_sphere*.
Precondition: if *min_sphere* is not empty, the dimension of *p* equals *ambient_dimension()*.

bool *min_sphere.is_empty() const*

returns *true*, iff *min_sphere* is empty (this implies degeneracy).

bool *min_sphere.is_degenerate() const*

returns *true*, iff *min_sphere* is degenerate, i.e. if *min_sphere* is empty or equal to a single point, equivalently if the number of support points is less than 2.

Modifiers

void *min_sphere.clear()*

resets *min_sphere* to $ms(\emptyset)$.

template < class InputIterator >

void *min_sphere.set(InputIterator first, InputIterator last)*

sets *min_sphere* to the $ms(P)$, where P is the set of points in the range [*first*,*last*).
Requirement: The value type of *first* and *last* is *Point*.
Precondition: All points have the same dimension.

void *min_sphere.insert(Point p)*

inserts p into *min_sphere*. If p lies inside the current sphere, this is a constant-time operation, otherwise it might take longer, but usually substantially less than recomputing the smallest enclosing sphere from scratch.
Precondition: The dimension of p equals *ambient_dimension()* if *min_sphere* is not empty.

template < class InputIterator >
void *min_sphere.insert(InputIterator first, InputIterator last)*

inserts the points in the range [*first*,*last*) into *min_sphere* and recomputes the smallest enclosing sphere, by calling *insert* for all points in the range.
Requirement: The value type of *first* and *last* is *Point*.
Precondition: All points have the same dimension. If *min_sphere* is not empty, this dimension must be equal to *ambient_dimension()*.

Validity Check

An object *min_sphere* is valid, iff

- *min_sphere* contains all points of its defining set P,
- *min_sphere* is the smallest sphere containing its support set S, and
- S is minimal, i.e. no support point is redundant.

Note: Under inexact arithmetic, the result of the validation is not realiable, because the checker itself can suffer from numerical problems.

bool *min_sphere.is_valid(bool verbose = false, int level = 0) const*

returns *true*, iff *min_sphere* is valid. If *verbose* is *true*, some messages concerning the performed checks are written to standard error stream. The second parameter *level* is not used, we provide it only for consistency with interfaces of other classes.

Miscellaneous

const Traits&　　*min_sphere.traits() const*

returns a const reference to the traits class object.

I/O

std::ostream&　　*std::ostream& os << min_sphere*

writes *min_sphere* to output stream *os*.
Requirement: The output operator is defined for *Point*.

std::istream&　　*std::istream& is >> min_sphere&*

reads *min_sphere* from input stream *is*.
Requirement: The input operator is defined for *Point*.

See Also

CGAL::Optimisation_d_traits_2<K,ET,NT> . . . . page **??**
CGAL::Optimisation_d_traits_3<K,ET,NT> . . . . page **??**
CGAL::Optimisation_d_traits_d<K,ET,NT> . . . . page **??**
OptimisationDTraits . . . . page **??**
CGAL::Min_circle_2<Traits> . . . . page 3802
CGAL::Min_sphere_of_spheres_d<Traits> . . . . page 3867
CGAL::Min_annulus_d<Traits> . . . . page 3860

Implementation

We implement the algorithm of Welzl with move-to-front heuristic [Wel91] for small point sets, combined with a new efficient method for large sets, which is particularly tuned for moderately large dimension ($d \leq 20$) [Gär99]. The creation time is almost always linear in the number of points. Access functions and predicates take constant time, inserting a point might take up to linear time, but substantially less than computing the new smallest enclosing sphere from scratch. The clear operation and the check for validity each take linear time.

Example

```
#include <CGAL/Cartesian_d.h>
#include <iostream>
#include <cstdlib>
#include <CGAL/Random.h>
#include <CGAL/Min_sphere_annulus_d_traits_d.h>
#include <CGAL/Min_sphere_d.h>

typedef CGAL::Cartesian_d<double>              K;
typedef CGAL::Min_sphere_annulus_d_traits_d<K> Traits;
typedef CGAL::Min_sphere_d<Traits>             Min_sphere;
typedef K::Point_d                             Point;

const int n = 10;                         // number of points
```

```
const int d = 5;                               // dimension of points

int main ()
{
    Point          P[n];                       // n points
    double         coord[d];                   // d coordinates
    CGAL::Random   r;                          // random number generator

    for (int i=0; i<n; ++i) {
        for (int j=0; j<d; ++j)
            coord[j] = r.get_double();
        P[i] = Point(d, coord, coord+d); // random point
    }

    Min_sphere  ms (P, P+n);                   // smallest enclosing sphere

    CGAL::set_pretty_mode (std::cout);
    std::cout << ms;                           // output the sphere

    return 0;
}
```

File: examples/Min_sphere_d/min_sphere_d.cpp

MinSphereAnnulusDTraits

Definition

This concept defines the requirements for traits classes of d-dimensional min sphere and min annulus algorithms.

Types

MinSphereAnnulusDTraits:: Point_d point type used to represent the input points.

MinSphereAnnulusDTraits:: Rep_tag compile time tag to distinguish between Cartesian and homogeneous representation of the input points. *Rep_tag* has to be either *CGAL::Cartesian_tag* or *CGAL::Homogeneous_tag*.

MinSphereAnnulusDTraits:: RT number type used to represent the coordinates of the input points. It has to be a model for *RingNumberType*.

MinSphereAnnulusDTraits:: FT number type used to return either the squared radius of the smallest enclosing sphere or annulus, or the squared distance of the polytopes. *FT* has to be either *RT* or *CGAL::Quotient*<*RT*> if the input points have Cartesian or homogeneous representation, respectively (cf. *Rep_tag*).

MinSphereAnnulusDTraits:: Access_dimension_d

data accessor object used to access the dimension of the input points.

MinSphereAnnulusDTraits:: Access_coordinates_begin_d

data accessor object used to access the coordinates of the input points.

MinSphereAnnulusDTraits:: Construct_point_d

constructor object used to construct either the center of the smallest enclosing sphere or annulus, or the points realizing the distance between the two polytopes.

The following two number types are only needed for *CGAL::Min_annulus_d*<*Traits*> and *CGAL::Polytope_distance_d*<*Traits*>.

MinSphereAnnulusDTraits:: ET exact number type used to do the exact computations in the underlying solver for linear programs. It has to to be a model for *RingNumberType*. There must be an implicit conversion from *RT* to *ET* available.

MinSphereAnnulusDTraits:: NT fast (possibly inexact) number type used to speed up the pricing step in the underlying solver for linear programs. It has to be a model for *RingNumberType*. There must be implicit conversions from *RT* to *NT* and from *NT* to *ET* available.

Creation

Only default and copy constructor are required.

MinSphereAnnulusDTraits traits;

MinSphereAnnulusDTraits traits(MinSphereAnnulusDTraits);

Operations

The following functions just return the corresponding function class object.

Access_dimension_d	*traits.access_dimension_d_object() const*
Access_coordinates_begin_d	*traits.access_coordinates_begin_d_object() const*
Construct_point_d	*traits.construct_point_d_object() const*

Has Models

CGAL::Min_sphere_annulus_d_traits_2<K,ET,NT> . . . . page 3901
CGAL::Min_sphere_annulus_d_traits_3<K,ET,NT> . . . . page 3903
CGAL::Min_sphere_annulus_d_traits_d<K,ET,NT> . . . . page 3905

See Also

CGAL::Min_sphere_d<Traits> . . . . page 3846
CGAL::Min_annulus_d<Traits> . . . . page 3860

Class

CGAL::Min_sphere_annulus_d_traits_2<K,ET,NT>

Definition

The class *Min_sphere_annulus_d_traits_2<K,ET,NT>* is a traits class for the d-dimensional optimisation algorithms using the two-dimensional CGAL kernel.

#include <CGAL/Min_sphere_annulus_d_traits_2.h>

Requirements

The template parameter *K* is a model for *Kernel*. Template parameters *ET* and *NT* are models for *RingNumberType*.

The second and third template parameter have default type *K::RT*.

Is Model for the Concepts

MinSphereAnnulusDTraits . . . . page 3907

Types

Min_sphere_annulus_d_traits_2<K,ET,NT>:: Point_d — typedef to *K::Point_2*.

Min_sphere_annulus_d_traits_2<K,ET,NT>:: Rep_tag — typedef to *K::Rep_tag*.

Min_sphere_annulus_d_traits_2<K,ET,NT>:: RT — typedef to *K::RT*.

Min_sphere_annulus_d_traits_2<K,ET,NT>:: FT — typedef to *K::FT*.

Min_sphere_annulus_d_traits_2<K,ET,NT>:: Access_dimension_d

typedef to *K::Access_dimension_2*.

Min_sphere_annulus_d_traits_2<K,ET,NT>:: Access_coordinates_begin_d

typedef to *K::Access_coordinates_begin_2*.

Min_sphere_annulus_d_traits_2<K,ET,NT>:: Construct_point_d

typedef to *K::Construct_point_2*.

Min_sphere_annulus_d_traits_2<K,ET,NT>:: ET — second template parameter (default is *K::RT*).

Min_sphere_annulus_d_traits_2<K,ET,NT>:: NT — third template parameter (default is *K::RT*).

Creation

Min_sphere_annulus_d_traits_2<K,ET,NT> traits; default constructor.

Min_sphere_annulus_d_traits_2<K,ET,NT> traits(Min_sphere_annulus_d_traits_2<K,ET,NT>);

copy constructor.

Operations

The following functions just return the corresponding function class object.

Access_dimension_d *traits.access_dimension_d_object() const*

Access_coordinates_begin_d *traits.access_coordinates_begin_d_object() const*

Construct_point_d *traits.construct_point_d_object() const*

See Also

CGAL::Min_sphere_d<Traits> . . . . page 3846
CGAL::Min_annulus_d<Traits> . . . . page 3860
CGAL::Min_sphere_annulus_d_traits_3<K,ET,NT> . . . . page 3903
CGAL::Min_sphere_annulus_d_traits_d<K,ET,NT> . . . . page 3905

MinSphereAnnulusDTraits . . . . page 3907

CGAL::Min_sphere_annulus_d_traits_3<K,ET,NT>

Definition

The class *Min_sphere_annulus_d_traits_3<K,ET,NT>* is a traits class for the d-dimensional optimisation algorithms using the three-dimensional CGAL kernel.

#include <CGAL/Min_sphere_annulus_d_traits_3.h>

Requirements

The template parameter *K* is a model for *Kernel*. Template parameters *ET* and *NT* are models for *RingNumberType*.

The second and third template parameter have default type *K::RT*.

Is Model for the Concepts

MinSphereAnnulusDTraits . . . . page 3907

Types

Min_sphere_annulus_d_traits_3<K,ET,NT>:: Point_d	typedef to *K::Point_3*.
Min_sphere_annulus_d_traits_3<K,ET,NT>:: Rep_tag	typedef to *K::Rep_tag*.
Min_sphere_annulus_d_traits_3<K,ET,NT>:: RT	typedef to *K::RT*.
Min_sphere_annulus_d_traits_3<K,ET,NT>:: FT	typedef to *K::FT*.
Min_sphere_annulus_d_traits_3<K,ET,NT>:: Access_dimension_d	typedef to *K::Access_dimension_3*.
Min_sphere_annulus_d_traits_3<K,ET,NT>:: Access_coordinates_begin_d	typedef to *K::Access_coordinates_begin_3*.
Min_sphere_annulus_d_traits_3<K,ET,NT>:: Construct_point_d	typedef to *K::Construct_point_3*.
Min_sphere_annulus_d_traits_3<K,ET,NT>:: ET	second template parameter (default is *K::RT*).
Min_sphere_annulus_d_traits_3<K,ET,NT>:: NT	third template parameter (default is *K::RT*).

Creation

Min_sphere_annulus_d_traits_3<K,ET,NT> traits; default constructor.

Min_sphere_annulus_d_traits_3<K,ET,NT> traits(Min_sphere_annulus_d_traits_3<K,ET,NT>);

copy constructor.

Operations

The following functions just return the corresponding function class object.

Access_dimension_d *traits.access_dimension_d_object() const*

Access_coordinates_begin_d *traits.access_coordinates_begin_d_object() const*

Construct_point_d *traits.construct_point_d_object() const*

See Also

CGAL::Min_sphere_d<Traits> . . . . page 3846
CGAL::Min_annulus_d<Traits> . . . . page 3860
CGAL::Min_sphere_annulus_d_traits_2<K,ET,NT> . . . . page 3901
CGAL::Min_sphere_annulus_d_traits_d<K,ET,NT> . . . . page 3905
Min_sphere_annulusDTraits . . . . page **??**

Class

CGAL::Min_sphere_annulus_d_traits_d<K,ET,NT>

Definition

The class *Min_sphere_annulus_d_traits_d<K,ET,NT>* is a traits class for the d-dimensional optimisation algorithms using the d-dimensional CGAL kernel.

#include <CGAL/Min_sphere_annulus_d_traits_d.h>

Requirements

The template parameter *K* is a model for *Kernel*. Template parameters *ET* and *NT* are models for *RingNumberType*.

The second and third template parameter have default type *K::RT*.

Is Model for the Concepts

MinSphereAnnulusDTraits . . . . page 3907

Types

Min_sphere_annulus_d_traits_d<K,ET,NT>:: Point_d	typedef to *K::Point_d*.
Min_sphere_annulus_d_traits_d<K,ET,NT>:: Rep_tag	typedef to *K::Rep_tag*.
Min_sphere_annulus_d_traits_d<K,ET,NT>:: RT	typedef to *K::RT*.
Min_sphere_annulus_d_traits_d<K,ET,NT>:: FT	typedef to *K::FT*.
Min_sphere_annulus_d_traits_d<K,ET,NT>:: Access_dimension_d	typedef to *K::Access_dimension_d*.
Min_sphere_annulus_d_traits_d<K,ET,NT>:: Access_coordinates_begin_d	typedef to *K::Access_coordinates_begin_d*.
Min_sphere_annulus_d_traits_d<K,ET,NT>:: Construct_point_d	typedef to *K::Construct_point_d*.
Min_sphere_annulus_d_traits_d<K,ET,NT>:: ET	second template parameter (default is *K::RT*).
Min_sphere_annulus_d_traits_d<K,ET,NT>:: NT	third template parameter (default is *K::RT*).

Creation

Min_sphere_annulus_d_traits_d<K,ET,NT> traits; default constructor.

Min_sphere_annulus_d_traits_d<K,ET,NT> traits(Min_sphere_annulus_d_traits_d<K,ET,NT>);

copy constructor.

Operations

The following functions just return the corresponding function class object.

Access_dimension_d	*traits.access_dimension_d_object() const*
Access_coordinates_begin_d	*traits.access_coordinates_begin_d_object() const*
Construct_point_d	*traits.construct_point_d_object() const*

See Also

CGAL::Min_sphere_d<Traits> . . . . page 3846
CGAL::Min_annulus_d<Traits> . . . . page 3860
CGAL::Polytope_distance_d<Traits> . . . . page 3944

CGAL::Min_sphere_annulus_d_traits_2<K,ET,NT> . . . . page 3901
CGAL::Min_sphere_annulus_d_traits_3<K,ET,NT> . . . . page 3903

MinSphereAnnulusDTraits . . . . page 3907

Class

CGAL::Min_annulus_d<Traits>

Definition

An object of the class *Min_annulus_d<Traits>* is the unique annulus (region between two concentric spheres with radii r and R, $r \leq R$) enclosing a finite set of points in d-dimensional Euclidean space $\mathbb{E}^d$, where the difference $R^2 - r^2$ is minimal. For a point set P we denote by $ma(P)$ the smallest annulus that contains all points of P. Note that $ma(P)$ can be degenerate, i.e. $ma(P) = \emptyset$ if $P = \emptyset$ and $ma(P) = \{p\}$ if $P = \{p\}$.

An inclusion-minimal subset S of P with $ma(S) = ma(P)$ is called a *support set*, the points in S are the *support points*. A support set has size at most $d+2$, and all its points lie on the boundary of $ma(P)$. In general, the support set is not necessarily unique.

The underlying algorithm can cope with all kinds of input, e.g. P may be empty or points may occur more than once. The algorithm computes a support set S which remains fixed until the next set, insert, or clear operation.

#include <CGAL/Min_annulus_d.h>

Requirements

The template parameter *Traits* is a model for *OptimisationDTraits*.

We provide the models *Optimisation_d_traits_2*, *Optimisation_d_traits_3*, and *Optimisation_d_traits_d* using the two-, three-, and d-dimensional CGAL kernel, respectively.

Types

Min_annulus_d<Traits>:: Point — typedef to *Traits::Point_d*. Point type used to represent the input points.

Min_annulus_d<Traits>:: FT — typedef to *Traits::FT*. Number type used to return the squared radii of the smallest enclosing annulus.

Min_annulus_d<Traits>:: ET — typedef to *Traits::ET*. Number type used to do the exact computations in the underlying solver for quadratic programs (cf. **Implementation**).

Min_annulus_d<Traits>:: Point_iterator

non-mutable model of the STL concept *RandomAccessIterator* with value type *Point*. Used to access the points of the smallest enclosing annulus.

Min_annulus_d<Traits>:: Support_point_iterator

non-mutable model of the STL concept *RandomAccessIterator* with value type *Point*. Used to access the support points of the smallest enclosing annulus.

Min_annulus_d<Traits>:: Inner_support_point_iterator

non-mutable model of the STL concept *RandomAccessIterator* with value type *Point*. Used to access the inner support points of the smallest enclosing annulus.

Min_annulus_d<Traits>:: Outer_support_point_iterator

non-mutable model of the STL concept *RandomAccessIterator* with value type *Point*. Used to access the outer support points of the smallest enclosing annulus.

Min_annulus_d<Traits>:: Coordinate_iterator

non-mutable model of the STL concept *RandomAccessIterator* with value type *ET*. Used to access the coordinates of the center of the smallest enclosing annulus.

Creation

Min_annulus_d<Traits> min_annulus(Traits traits = Traits(), int verbose = 0, std::ostream& stream = std::cout)

initializes *min_annulus* to $ma(\emptyset)$.

template < class InputIterator >
Min_annulus_d<Traits> min_annulus(InputIterator first, InputIterator last, Traits traits = Traits(), int verbose = 0, std::ostream& stream = std::cout)

initializes *min_annulus* to $ma(P)$ with P being the set of points in the range [*first*,*last*).
Requirement: The value type of *InputIterator* is *Point*.
Precondition: All points have the same dimension.

Access Functions

int min_annulus.ambient_dimension() const

returns the dimension of the points in P. If *min_annulus* is empty, the ambient dimension is -1.

int min_annulus.number_of_points() const

returns the number of points of *min_annulus*, i.e. $|P|$.

int min_annulus.number_of_support_points() const

returns the number of support points of *min_annulus*, i.e. $|S|$.

int *min_annulus.number_of_inner_support_points() const*

returns the number of support points of *min_annulus* which lie on the inner sphere.

int *min_annulus.number_of_outer_support_points() const*

returns the number of support points of *min_annulus* which lie on the outer sphere.

Point_iterator
min_annulus.points_begin() const
returns an iterator referring to the first point of *min_annulus*.

Point_iterator
min_annulus.points_end() const
returns the corresponding past-the-end iterator.

Support_point_iterator
min_annulus.support_points_begin() const
returns an iterator referring to the first support point of *min_annulus*.
Precondition: $ma(P)$ is not degenerate, i.e., *number_of_support_points()* is at least one.

Support_point_iterator
min_annulus.support_points_end() const
returns the corresponding past-the-end iterator.
Precondition: $ma(P)$ is not degenerate, i.e., *number_of_support_points()* is at least one.

Inner_support_point_iterator
min_annulus.inner_support_points_begin() const
returns an iterator referring to the first inner support point of *min_annulus*.

Inner_support_point_iterator
min_annulus.inner_support_points_end() const
returns the corresponding past-the-end iterator.

Outer_support_point_iterator
min_annulus.outer_support_points_begin() const
returns an iterator referring to the first outer support point of *min_annulus*.

Outer_support_point_iterator
min_annulus.outer_support_points_end() const
returns the corresponding past-the-end iterator.

Point *min_annulus.center() const*
returns the center of *min_annulus*.
Requirement: An implicit conversion from *ET* to *RT* is available.
Precondition: *min_annulus* is not empty.

FT *min_annulus.squared_inner_radius() const*

returns the squared inner radius of *min_annulus*.
Requirement: An implicit conversion from *ET* to *RT* is available.
Precondition: *min_annulus* is not empty.

FT *min_annulus.squared_outer_radius() const*

returns the squared outer radius of *min_annulus*.
Requirement: An implicit conversion from *ET* to *RT* is available.
Precondition: *min_annulus* is not empty.

Coordinate_iterator
min_annulus.center_coordinates_begin() const
returns an iterator referring to the first coordinate of the center of *min_annulus*.
Note: The coordinates have a rational representation, i.e. the first d elements of the iterator range are the numerators and the $(d+1)$-st element is the common denominator.

Coordinate_iterator
min_annulus.center_coordinates_end() const
returns the corresponding past-the-end iterator.

ET *min_annulus.squared_inner_radius_numerator() const*

returns the numerator of the squared inner radius of *min_annulus*.

ET *min_annulus.squared_outer_radius_numerator() const*

returns the numerator of the squared outer radius of *min_annulus*.

ET *min_annulus.squared_radii_denominator() const*

returns the denominator of the squared radii of *min_annulus*.

Predicates

The bounded area of the smallest enclosing annulus lies between the inner and the outer sphere. The boundary is the union of both spheres. By definition, an empty annulus has no boundary and no bounded side, i.e. its unbounded side equals the whole space $\mathbb{E}^d$.

CGAL::Bounded_side

min_annulus.bounded_side(Point p) const

returns *CGAL::ON_BOUNDED_SIDE*, *CGAL::ON_BOUNDARY*, or *CGAL::ON_UNBOUNDED_SIDE* iff p lies properly inside, on the boundary, or properly outside of *min_annulus*, resp.
Precondition: The dimension of p equals *min_annulus.ambient_dimension()* if *min_annulus* is not empty.

bool *min_annulus.has_on_bounded_side(Point p) const*

returns *true*, iff p lies properly inside *min_annulus*.
Precondition: The dimension of p equals *min_annulus.ambient_dimension()* if *min_annulus* is not empty.

bool *min_annulus.has_on_boundary(Point p) const*

returns *true*, iff *p* lies on the boundary of *min_annulus*.
Precondition: The dimension of *p* equals *min_annulus.ambient_dimension()* if *min_annulus* is not empty.

bool *min_annulus.has_on_unbounded_side(Point p) const*

returns *true*, iff *p* lies properly outside of *min_annulus*.
Precondition: The dimension of *p* equals *min_annulus.ambient_dimension()* if *min_annulus* is not empty.

bool *min_annulus.is_empty() const*

returns *true*, iff *min_annulus* is empty (this implies degeneracy).

bool *min_annulus.is_degenerate() const*

returns *true*, iff *min_annulus* is degenerate, i.e. if *min_annulus* is empty or equal to a single point.

Modifiers

void *min_annulus.clear()* resets *min_annulus* to $ma(\emptyset)$.

template < class InputIterator >
void *min_annulus.set(InputIterator first, InputIterator last)*

sets *min_annulus* to $ma(P)$, where P is the set of points in the range [*first*,*last*).
Requirement: The value type of *InputIterator* is *Point*.
Precondition: All points have the same dimension.

void *min_annulus.insert(Point p)*

inserts *p* into *min_annulus*.
Precondition: The dimension of *p* equals *min_annulus.ambient_dimension()* if *min_annulus* is not empty.

template < class InputIterator >
void *min_annulus.insert(InputIterator first, InputIterator last)*

inserts the points in the range [*first*,*last*) into *min_annulus* and recomputes the smallest enclosing annulus.
Requirement: The value type of *InputIterator* is *Point*.
Precondition: All points have the same dimension. If *min_annulus* is not empty, this dimension must be equal to *min_annulus.ambient_dimension()*.

Validity Check

An object *min_annulus* is valid, iff

- *min_annulus* contains all points of its defining set *P*,
- *min_annulus* is the smallest annulus containing its support set *S*, and
- *S* is minimal, i.e. no support point is redundant.

Note: In this release only the first item is considered by the validity check.

bool *min_annulus.is_valid(bool verbose = false, int level = 0) const*

returns *true*, iff *min_annulus* is valid. If *verbose* is *true*, some messages concerning the performed checks are written to standard error stream. The second parameter *level* is not used, we provide it only for consistency with interfaces of other classes.

Miscellaneous

const Traits&

min_annulus.traits() const returns a const reference to the traits class object.

I/O

std::ostream&

std::ostream& os << *min_annulus*

writes *min_annulus* to output stream *os*.
Requirement: The output operator is defined for *Point*.

std::istream&

std::istream& is >> *min_annulus&*

reads *min_annulus* from input stream *is*.
Requirement: The input operator is defined for *Point*.

See Also

CGAL::Min_sphere_d<Traits> . . . . page 3846

CGAL::Optimisation_d_traits_2<K,ET,NT> . . . . page **??**
CGAL::Optimisation_d_traits_3<K,ET,NT> . . . . page **??**
CGAL::Optimisation_d_traits_d<K,ET,NT> . . . . page **??**

OptimisationDTraits . . . . page **??**

Implementation

The problem of finding the smallest enclosing annulus of a finite point set can be formulated as an optimization problem with linear constraints and a linear objective function. The solution is obtained using our exact solver for linear and quadratic programs [GS00].

The creation time is almost always linear in the number of points. Access functions and predicates take constant time, inserting a point takes almost always linear time. The clear operation and the check for validity each take linear time.

CGAL::Min_sphere_of_spheres_d<Traits>

Class

Definition

An object of the class *Min_sphere_of_spheres_d<Traits>* is a data structure that represents the unique sphere of smallest volume enclosing a finite set of spheres in d-dimensional Euclidean space $\mathbb{E}^d$. For a set S of spheres we denote by $ms(S)$ the smallest sphere that contains all spheres of S; we call $ms(S)$ the *minsphere* of S. $ms(S)$ can be degenerate, i.e., $ms(S) = \emptyset$, if $S = \emptyset$ and $ms(S) = \{s\}$, if $S = \{s\}$. Any sphere in S may be degenerate, too, i.e., any sphere from S may be a point. Also, S may contain several copies of the same sphere.

An inclusion-minimal subset R of S with $ms(R) = ms(S)$ is called a *support set* for $ms(S)$; the spheres in R are the *support spheres*. A support set has size at most $d+1$, and all its spheres lie on the boundary of $ms(S)$. (A sphere s' is said to *lie on the boundary* of a sphere s, if s' is contained in s and if their boundaries intersect.) In general, the support set is not unique.

The algorithm computes the center and the radius of $ms(S)$, and finds a support set R (which remains fixed until the next *insert()*, *clear()* or *set()* operation). We also provide a specialization of the algorithm for the case when the center coordinates and radii of the input spheres are floating-point numbers. This specialized algorithm uses floating-point arithmetic only, is very fast and especially tuned for stability and robustness. Still, it's output may be incorrect in some (rare) cases; termination is guaranteed.

When default constructed, an instance of type *Min_sphere_of_spheres_d<Traits>* represents the set $S = \emptyset$, together with its minsphere $ms(S) = \emptyset$. You can add spheres to the set S by calling *insert()*. Querying the minsphere is done by calling the routines *is_empty()*, *radius()* and *center_cartesian_begin()*, among others.

In general, the radius and the Euclidean center coordinates of $ms(S)$ need not be rational. Consequently, the algorithm computing the exact minsphere will have to deal with algebraic numbers. Fortunately, both the radius and the coordinates of the minsphere are numbers of the form $a_i + b_i\sqrt{t}$, where $a_i, b_i, t \in \mathbb{Q}$ and where $t \geq 0$ is the same for all coordinates and the radius. Thus, the exact minsphere can be described by the number t, which is called the sphere's *discriminant*, and by $d+1$ pairs $(a_i, b_i) \in \mathbb{Q}^2$ (one for the radius and d for the center coordinates).

#include <CGAL/Min_sphere_of_spheres_d.h>

Note: This class (almost) replaces *CGAL::Min_sphere_d<Traits>*, which solves the less general problem of finding the smallest enclosing ball of a set of *points*. *Min_sphere_of_spheres_d<Traits>* is faster than *CGAL::Min_sphere_d<Traits>*, and in contrast to the latter provides a specialized implementation for floating-point arithmetic which ensures correct results in a large number of cases (including highly degenerate ones). The only advantage of *CGAL::Min_sphere_d<Traits>* over *Min_sphere_of_spheres_d<Traits>* is that the former can deal with points in homogeneous coordinates, in which case the algorithm is division-free. Thus, *CGAL::Min_sphere_d<Traits>* might still be an option in case your input number type cannot (efficiently) divide.

Requirements

The class *Min_sphere_of_spheres_d<Traits>* expects a model of the concept *MinSphereOfSpheresTraits* as its template argument.

Types

Min_sphere_of_spheres_d<Traits>:: Sphere is a typedef to *Traits::Sphere*.

Min_sphere_of_spheres_d<Traits>:: FT is a typedef to *Traits::FT*.

Min_sphere_of_spheres_d<Traits>:: Result is the type of the radius and of the center coordinates of the computed minsphere: When *FT* is an inexact number type (*double*, for instance), then *Result* is simply *FT*. However, when *FT* is an exact number type, then *Result* is a typedef to a derived class of *std::pair<FT,FT>*; an instance of this type represents the number $a+b\sqrt{t}$, where a is the first and b the second element of the pair and where the number t is accessed using the member function *discriminant()* of class *Min_sphere_of_spheres_d<Traits>*.

Min_sphere_of_spheres_d<Traits>:: Algorithm is either *CGAL::LP_algorithm* or *CGAL::Farthest_first_heuristic*. As is described in the documentation of concept *MinSphereOfSpheresTraits*, the type *Algorithm* reflects the method which is used to compute the minsphere. (Normally, *Algorithm* coincides with *Traits::Algorithm*. However, if the method *Traits::Algorithm* should not be supported anymore in a future release, then *Algorithm* will have another type.)

Min_sphere_of_spheres_d<Traits>:: Support_iterator

non-mutable model of the STL concept *BidirectionalIterator* with value type *Sphere*. Used to access the support spheres defining the smallest enclosing sphere.

Min_sphere_of_spheres_d<Traits>:: Cartesian_const_iterator

non-mutable model of the STL concept *BidirectionalIterator* to access the center coordinates of the minsphere.

Creation

Min_sphere_of_spheres_d<Traits> minsphere(Traits traits = Traits());

creates a variable of type *Min_sphere_of_spheres_d<Traits>* and initializes it to $ms(\emptyset)$. If the traits parameter is not supplied, the class *Traits* must provide a default constructor.

template < typename InputIterator >
Min_sphere_of_spheres_d<Traits> minsphere(InputIterator first, InputIterator last, Traits traits = Traits());

creates a variable *minsphere* of type *Min_sphere_of_spheres_d<Traits>* and inserts (cf. *insert()*) the spheres from the range [*first*,*last*).
Requirement: The value type of *first* and *last* is *Sphere*. If the traits parameter is not supplied, the class *Traits* must provide a default constructor.

Access Functions

Support_iterator *minsphere.support_begin() const*

returns an iterator referring to the first support sphere of *minsphere*.

Support_iterator *minsphere.support_end() const*

returns the corresponding past-the-end iterator.

FT *minsphere.discriminant() const*

returns the discriminant of *minsphere*. This number is undefined when *FT* is an inexact number type. When *FT* is exact, the center coordinates and the radius of the minsphere are numbers of the form $a+b\sqrt{t}$, where t is the discriminant of the minsphere as returned by this function.
Precondition: *minsphere* is not empty, and *FT* is an exact number type.

Result *minsphere.radius() const*

returns the radius of *minsphere*. If *FT* is an exact number type then the radius of the minsphere is the real number $a+b\sqrt{t}$, where t is the minsphere's discriminant, a is the first and b the second component of the pair returned by *radius()*.
Precondition: *minsphere* is not empty.

Cartesian_const_iterator

minsphere.center_cartesian_begin() const

returns a const-iterator to the first of the *Traits::D* center coordinates of *minsphere*. The iterator returns objects of type *Result*. If *FT* is an exact number type, then a center coordinate is represented by a pair (a,b) describing the real number $a+b\sqrt{t}$, where t is the minsphere's discriminant (cf. *discriminant()*).
Precondition: *minsphere* is not empty.

Cartesian_const_iterator

minsphere.center_cartesian_end() const

returns the corresponding past-the-end iterator, i.e. *center_cartesian_begin()+Traits::D*.
Precondition: *minsphere* is not empty.

Predicates

bool *minsphere.is_empty() const*

returns *true*, iff *minsphere* is empty, i.e. iff $ms(S) = \emptyset$.

Modifiers

void *minsphere.clear()* resets *minsphere* to $ms(\emptyset)$, with $S := \emptyset$.

template < class InputIterator >
void *minsphere.set(InputIterator first, InputIterator last)*

sets *minsphere* to the $ms(S)$, where S is the set of spheres in the range [*first*,*last*).
Requirement: The value type of *first* and *last* is *Sphere*.

void *minsphere.insert(Sphere s)*

inserts the sphere *s* into the set S of instance *minsphere*.

template < class InputIterator >
void *minsphere.insert(InputIterator first, InputIterator last)*

inserts the spheres in the range [*first*,*last*) into the set S of instance *minsphere*.
Requirement: The value type of *first* and *last* is *Sphere*.

Validity Check

An object *minsphere* is valid, iff

- *minsphere* contains all spheres of its defining set S,
- *minsphere* is the smallest sphere containing its support set R, and
- R is minimal, i.e., no support sphere is redundant.

bool *minsphere.is_valid() const*

returns *true*, iff *minsphere* is valid. When *FT* is inexact, this routine always returns *true*.

Miscellaneous

const Traits& *minsphere.traits() const*

returns a const reference to the traits class object.

See Also

CGAL::Min_sphere_d<Traits> . . . page 3846
CGAL::Min_circle_2<Traits> . . . page 3802

Implementation

We implement two algorithms, the LP-algorithm and a heuristic [MSW92]. As described in the documentation of concept *MinSphereOfSpheresTraits*, each has its advantages and disadvantages: Our implementation of the LP-algorithm has maximal expected running time $O(2^d n)$, while the heuristic comes without any complexity guarantee. In particular, the LP-algorithm runs in linear time for fixed dimension d. (These running times hold for the arithmetic model, so they count the number of operations on the number type *FT*.)

On the other hand, the LP-algorithm is, for inexact number types *FT*, much worse at handling degeneracies and should therefore not be used in such a case. (For exact number types *FT*, both methods handle all kinds of degeneracies.)

Currently, we require *Traits::FT* to be either an exact number type or *double* or *float*; other inexact number types are not supported at this time. Also, the current implementation only handles spheres with Cartesian coordinates; homogenous representation is not supported yet.

Example

```
#include <CGAL/Cartesian_d.h>
#include <CGAL/Random.h>
#include <CGAL/Gmpq.h>
#include <CGAL/Min_sphere_of_spheres_d.h>
#include <vector>

const int N = 1000;                       // number of spheres
const int D = 3;                          // dimension of points
const int LOW = 0, HIGH = 10000;          // range of coordinates and
    radii

typedef CGAL::Gmpq                        FT;
//typedef double                          FT;
typedef CGAL::Cartesian_d<FT>             K;
typedef CGAL::Min_sphere_of_spheres_d_traits_d<K,FT,D> Traits;
typedef CGAL::Min_sphere_of_spheres_d<Traits> Min_sphere;
typedef K::Point_d                        Point;
typedef Traits::Sphere                    Sphere;

int main () {
  std::vector<Sphere> S;                  // n spheres
  FT coord[D];                            // d coordinates
  CGAL::Random r;                         // random number generator

  for (int i=0; i<N; ++i) {
    for (int j=0; j<D; ++j)
      coord[j] = r.get_int(LOW,HIGH);
    Point p(D,coord,coord+D);             // random center...
    S.push_back(Sphere(p,r.get_int(LOW,HIGH))); // ...and random radius
```

```
  }

  Min_sphere ms(S.begin(),S.end());       // check in the spheres
  CGAL_assertion(ms.is_valid());
}
```

File: examples/Min_sphere_of_spheres_d/min_sphere_of_spheres_d_d.cpp

MinSphereOfSpheresTraits

Definition

A model of concept *MinSphereOfSpheresTraits* must provide the following constants, types, predicates and operations.

Has Models

CGAL::Min_sphere_of_spheres_d_traits_2<K,FT,UseSqrt,Algorithm>
CGAL::Min_sphere_of_spheres_d_traits_3<K,FT,UseSqrt,Algorithm>
CGAL::Min_sphere_of_spheres_d_traits_d<K,FT,Dim,UseSqrt,Algorithm>

Constants

MinSphereOfSpheresTraits:: D — specifies the dimension of the spheres you want to compute the minsphere of.

Types

MinSphereOfSpheresTraits:: Sphere — is a typedef to to some class representing a sphere. (The package will compute the minsphere of spheres of type *Sphere*.) The type *Sphere* must provide a copy constructor.

MinSphereOfSpheresTraits:: FT — is a (exact or inexact) field number type.
Requirement: Currently, *FT* must either be *double* or *float*, or an exact field number type. (An *exact* number type is one which evaluates arithmetic expressions involving the four basic operations and comparisions with infinite precision, that is, like in $\mathbb{R}$.)

MinSphereOfSpheresTraits:: Cartesian_const_iterator

non-mutable model of the STL concept *ForwardIterator* with value type *FT*. Used to access the center coordinates of a sphere.

MinSphereOfSpheresTraits:: Use_square_roots

must typedef to either *CGAL::Tag_true* or *CGAL::Tag_false*. The algorithm uses (depending on the type *MinSphereOfSpheresTraits::Algorithm*) floating-point arithmetic internally for some intermediate computations. The type *Use_square_roots* affects how these calculations are done: When *Use_square_roots* is *Tag_true*, the algorithm computing the minsphere will perform square-root operations on *double*s and *float*s where appropriate. On the other hand, if *Use_square_roots* is *CGAL::Tag_false*, the algorithm will work without doing square-roots.
Note: On some platforms the algorithm is much faster when square-roots are disabled (due to lacking hardware support).

MinSphereOfSpheresTraits:: Algorithm

selects the method to compute the minsphere with. It must typedef to either *CGAL::Default_algorithm*, *CGAL::LP_algorithm* or *CGAL::Farthest_first_heuristic*. The recommended choice is the first, which is a synonym to the one of the other two methods which we consider "the best in practice." In case of *CGAL::LP_algorithm*, the minsphere will be computed using the LP-algorithm [MSW92], which in our implementation has maximal expected running time $O(2^d n)$ (in the number of operations on the number type *FT*). In case of *CGAL::Farthest_first_heuristic*, a simple heuristic will be used instead which seems to work fine in practice, but comes without a guarantee on the running time. For an inexact number type *FT* we strongly recommend *CGAL::Default_algorithm*, or, if you want, *CGAL::Farthest_first_heuristic*, since these handle most degeneracies in a satisfying manner. Notice that this compile-time flag is taken as a hint only. Should one of the methods not be available anymore in a future release, then the default algorithm will be chosen.

Access Functions

FT *traits.radius(Sphere s)*

returns the radius of sphere *s*.
Postcondition: The returned number is greater or equal to 0.

Cartesian_const_iterator

traits.center_cartesian_begin(Sphere s)

returns an iterator referring to the first of the *D* Cartesian coordinates of the center of *s*.

CGAL::Min_sphere_of_spheres_d_traits_2<K,FT,UseSqrt,Algorithm>

Definition

The class *Min_sphere_of_spheres_d_traits_2<K,FT,UseSqrt,Algorithm>* is a model for concept *MinSphereOfSpheresTraits*. It uses the CGAL type *Point_2* to represent circles.

Is Model for the Concepts

MinSphereOfSpheresTraits

Parameters

The last two template parameters, *UseSqrt* and *Algorithm*, have default arguments, namely *CGAL::Tag_false* and *CGAL::Default_algorithm*, respectively.

The template parameters of class *Min_sphere_of_spheres_d_traits_2<K,FT,UseSqrt,Algorithm>* must fulfill the following requirements:

Min_sphere_of_spheres_d_traits_2<K,FT,UseSqrt,Algorithm>:: K

is a model for *Kernel*.

Min_sphere_of_spheres_d_traits_2<K,FT,UseSqrt,Algorithm>:: FT

is a number type, which fulfills the requirements of type *FT* of concept *MinSphereOfSpheresTraits*: It must be either *double* or *float*, or an exact number type.

Min_sphere_of_spheres_d_traits_2<K,FT,UseSqrt,Algorithm>:: UseSqrt

fulfills the requirements of type *Use_square_roots* of concept *MinSphereOfSpheresTraits*: It must be either *Tag_true* or *Tag_false*.

Min_sphere_of_spheres_d_traits_2<K,FT,UseSqrt,Algorithm>:: Algorithm

fulfills the requirements of type *Algorithm* of concept *MinSphereOfSpheresTraits*: It must be either *Default_algorithm*, *LP_algorithm* or *Farthest_first_heuristic*.

Constants

Min_sphere_of_spheres_d_traits_2<K,FT,UseSqrt,Algorithm>:: D

is the constant 2, i.e. the dimension of $\mathbb{R}^2$.

Types

In addition to the types required by the concept *MinSphereOfSpheresTraits*, this model also defines the types *Radius* and *Point*. Here's the complete list of defined types:

Min_sphere_of_spheres_d_traits_2<K,FT,UseSqrt,Algorithm>:: FT

Min_sphere_of_spheres_d_traits_2<K,FT,UseSqrt,Algorithm>:: Use_square_roots

Min_sphere_of_spheres_d_traits_2<K,FT,UseSqrt,Algorithm>:: Algorithm

Min_sphere_of_spheres_d_traits_2<K,FT,UseSqrt,Algorithm>:: Radius

is a typedef to the template parameter *FT*

Min_sphere_of_spheres_d_traits_2<K,FT,UseSqrt,Algorithm>:: Point

is a typedef to *K::Point_2*.

Min_sphere_of_spheres_d_traits_2<K,FT,UseSqrt,Algorithm>:: Sphere

is a typedef to *std::pair<Point,Radius>*.

Min_sphere_of_spheres_d_traits_2<K,FT,UseSqrt,Algorithm>:: Cartesian_const_iterator

is a typedef to *K::Cartesian_const_iterator_2*.

Access Functions

The class provides the access functions required by the concept *MinSphereOfSpheresTraits*; they simply map to the corresponding routines of class *K::Point_2*:

FT *traits.radius(Sphere s)*

maps to *s.second*.

Cartesian_const_iterator

traits.center_cartesian_begin(Sphere s)

maps to *s.first.cartesian_begin()*.

CGAL::Min_sphere_of_spheres_d_traits_3<K,FT,UseSqrt,Algorithm>

Class

Definition

The class *Min_sphere_of_spheres_d_traits_3<K,FT,UseSqrt,Algorithm>* is a model for concept *MinSphereOfSpheresTraits*. It uses the CGAL type *Point_3* to represent circles.

Is Model for the Concepts

MinSphereOfSpheresTraits

Parameters

The last two template parameters, *UseSqrt* and *Algorithm*, have default arguments, namely *CGAL::Tag_false* and *CGAL::Default_algorithm*, respectively.

The template parameters of class *Min_sphere_of_spheres_d_traits_3<K,FT,UseSqrt,Algorithm>* must fulfill the following requirements:

Min_sphere_of_spheres_d_traits_3<K,FT,UseSqrt,Algorithm>:: K

is a model for *Kernel*.

Min_sphere_of_spheres_d_traits_3<K,FT,UseSqrt,Algorithm>:: FT

is a number type, which fulfills the requirements of type *FT* of concept *MinSphereOfSpheresTraits*: It must be either *double* or *float*, or an exact number type.

Min_sphere_of_spheres_d_traits_3<K,FT,UseSqrt,Algorithm>:: UseSqrt

fulfills the requirements of type *Use_square_roots* of concept *MinSphereOfSpheresTraits*: It must be either *Tag_true* or *Tag_false*.

Min_sphere_of_spheres_d_traits_3<K,FT,UseSqrt,Algorithm>:: Algorithm

fulfills the requirements of type *Algorithm* of concept *MinSphereOfSpheresTraits*: It must be either *Default_algorithm*, *LP_algorithm* or *Farthest_first_heuristic*.

Constants

Min_sphere_of_spheres_d_traits_3<K,FT,UseSqrt,Algorithm>:: D

is the constant 3, i.e. the dimension of $\mathbb{R}^3$.

Types

In addition to the types required by the concept *MinSphereOfSpheresTraits*, this model also defines the types *Radius* and *Point*. Here's the complete list of defined types:

Min_sphere_of_spheres_d_traits_3<K,FT,UseSqrt,Algorithm>:: FT

Min_sphere_of_spheres_d_traits_3<K,FT,UseSqrt,Algorithm>:: Use_square_roots

Min_sphere_of_spheres_d_traits_3<K,FT,UseSqrt,Algorithm>:: Algorithm

Min_sphere_of_spheres_d_traits_3<K,FT,UseSqrt,Algorithm>:: Radius

is a typedef to the template parameter *FT*

Min_sphere_of_spheres_d_traits_3<K,FT,UseSqrt,Algorithm>:: Point

is a typedef to *K::Point_3*.

Min_sphere_of_spheres_d_traits_3<K,FT,UseSqrt,Algorithm>:: Sphere

is a typedef to *std::pair<Point,Radius>*.

Min_sphere_of_spheres_d_traits_3<K,FT,UseSqrt,Algorithm>:: Cartesian_const_iterator

is a typedef to *K::Cartesian_const_iterator_2*.

Access Functions

The class provides the access functions required by the concept *MinSphereOfSpheresTraits*; they simply map to the corresponding routines of class *K::Point_3*:

FT *traits.radius(Sphere s)*

maps to *s.second*.

Cartesian_const_iterator

traits.center_cartesian_begin(Sphere s)

maps to *s.first.cartesian_begin()*.

CGAL::Min_sphere_of_spheres_d_traits_d<K,FT,Dim,UseSqrt,Algorithm>

Class

Definition

The class *Min_sphere_of_spheres_d_traits_d<K,FT,Dim,UseSqrt,Algorithm>* is a model for concept *MinSphereOfSpheresTraits*. It uses the CGAL type *Point_d* to represent circles.

Is Model for the Concepts

MinSphereOfSpheresTraits

Parameters

The last two template parameters, *UseSqrt* and *Algorithm*, have default arguments, namely *CGAL::Tag_false* and *CGAL::Default_algorithm*, respectively.

The template parameters of class *Min_sphere_of_spheres_d_traits_d<K,FT,UseSqrt,Algorithm>* must fulfill the following requirements:

Min_sphere_of_spheres_d_traits_d<K,FT,Dim,UseSqrt,Algorithm>:: K

is a model for *Kernel*.

Min_sphere_of_spheres_d_traits_d<K,FT,Dim,UseSqrt,Algorithm>:: FT

is a number type, which fulfills the requirements of type *FT* of concept *MinSphereOfSpheresTraits*: It must be either *double* or *float*, or an exact number type.

Min_sphere_of_spheres_d_traits_d<K,FT,Dim,UseSqrt,Algorithm>:: UseSqrt

fulfills the requirements of type *Use_square_roots* of concept *MinSphereOfSpheresTraits*: It must be either *Tag_true* or *Tag_false*.

Min_sphere_of_spheres_d_traits_d<K,FT,Dim,UseSqrt,Algorithm>:: Algorithm

fulfills the requirements of type *Algorithm* of concept *MinSphereOfSpheresTraits*: It must be either *Default_algorithm*, *LP_algorithm* or *Farthest_first_heuristic*.

Constants

Min_sphere_of_spheres_d_traits_d<K,FT,Dim,UseSqrt,Algorithm>:: D

is the constant *Dim*.

Types

In addition to the types required by the concept *MinSphereOfSpheresTraits*, this model also defines the types *Radius* and *Point*. Here's the complete list of defined types:

Min_sphere_of_spheres_d_traits_d<K,FT,Dim,UseSqrt,Algorithm>:: FT

Min_sphere_of_spheres_d_traits_d<K,FT,Dim,UseSqrt,Algorithm>:: Use_square_roots

Min_sphere_of_spheres_d_traits_d<K,FT,Dim,UseSqrt,Algorithm>:: Algorithm

Min_sphere_of_spheres_d_traits_d<K,FT,Dim,UseSqrt,Algorithm>:: Radius

is a typedef to the template parameter *FT*

Min_sphere_of_spheres_d_traits_d<K,FT,Dim,UseSqrt,Algorithm>:: Point

is a typedef to *K::Point_d*.

Min_sphere_of_spheres_d_traits_d<K,FT,Dim,UseSqrt,Algorithm>:: Sphere

is a typedef to *std::pair<Point,Radius>*.

Min_sphere_of_spheres_d_traits_d<K,FT,Dim,UseSqrt,Algorithm>:: Cartesian_const_iterator

is a typedef to *K::Cartesian_const_iterator_d*.

Access Functions

The class provides the access functions required by the concept *MinSphereOfSpheresTraits*; they simply map to the corresponding routines of class *K::Point_d*:

FT *traits.radius(Sphere s)*

maps to *s.second*.

Cartesian_const_iterator

traits.center_cartesian_begin(Sphere s)

maps to *s.first.cartesian_begin()*.

CGAL::Min_sphere_of_points_d_traits_2<K,FT,UseSqrt,Algorithm>

Definition

The class *Min_sphere_of_points_d_traits_2<K,FT,UseSqrt,Algorithm>* is a model for concept *MinSphereOfSpheresTraits*. It uses the CGAL type *Point_2* to represent circles.

Is Model for the Concepts

MinSphereOfSpheresTraits

Parameters

The last two template parameters, *UseSqrt* and *Algorithm*, have default arguments, namely *CGAL::Tag_false* and *CGAL::Default_algorithm*, respectively.

The template parameters of class *Min_sphere_of_points_d_traits_2<K,FT,UseSqrt,Algorithm>* must fulfill the following requirements:

Min_sphere_of_points_d_traits_2<K,FT,UseSqrt,Algorithm>:: K

is a model for *Kernel*.

Min_sphere_of_points_d_traits_2<K,FT,UseSqrt,Algorithm>:: FT

is a number type, which fulfills the requirements of type *FT* of concept *MinSphereOfSpheresTraits*: It must be either *double* or *float*, or an exact number type.

Min_sphere_of_points_d_traits_2<K,FT,UseSqrt,Algorithm>:: UseSqrt

fulfills the requirements of type *Use_square_roots* of concept *MinSphereOfSpheresTraits*: It must be either *Tag_true* or *Tag_false*.

Min_sphere_of_points_d_traits_2<K,FT,UseSqrt,Algorithm>:: Algorithm

fulfills the requirements of type *Algorithm* of concept *MinSphereOfSpheresTraits*: It must be either *Default_algorithm*, *LP_algorithm* or *Farthest_first_heuristic*.

Constants

Min_sphere_of_points_d_traits_2<K,FT,UseSqrt,Algorithm>:: D

is the constant 2, i.e. the dimension of $\mathbb{R}^2$.

Types

In addition to the types required by the concept *MinSphereOfSpheresTraits*, this model also defines the types *Radius* and *Point*. Here's the complete list of defined types:

Min_sphere_of_points_d_traits_2<K,FT,UseSqrt,Algorithm>:: FT

Min_sphere_of_points_d_traits_2<K,FT,UseSqrt,Algorithm>:: Use_square_roots

Min_sphere_of_points_d_traits_2<K,FT,UseSqrt,Algorithm>:: Algorithm

Min_sphere_of_points_d_traits_2<K,FT,UseSqrt,Algorithm>:: Radius

is a typedef to the template parameter *FT*

Min_sphere_of_points_d_traits_2<K,FT,UseSqrt,Algorithm>:: Point

is a typedef to *K::Point_2*.

Min_sphere_of_points_d_traits_2<K,FT,UseSqrt,Algorithm>:: Sphere

is a typedef to *Point*.

Min_sphere_of_points_d_traits_2<K,FT,UseSqrt,Algorithm>:: Cartesian_const_iterator

is a typedef to *K::Cartesian_const_iterator_2*.

Access Functions

The class provides the access functions required by the concept *MinSphereOfSpheresTraits*; they simply map to the corresponding routines of class *K::Point_2*:

FT *traits.radius(Sphere s)*

returns *0*.

Cartesian_const_iterator

traits.center_cartesian_begin(Sphere s)

maps to *s.first.cartesian_begin()*.

CGAL::Min_sphere_of_spheres_d_traits_3<K,FT,UseSqrt,Algorithm>

Definition

The class *Min_sphere_of_points_d_traits_3<K,FT,UseSqrt,Algorithm>* is a model for concept *MinSphereOfSpheresTraits*. It uses the CGAL type *Point_3* to represent circles.

Is Model for the Concepts

MinSphereOfSpheresTraits

Parameters

The last two template parameters, *UseSqrt* and *Algorithm*, have default arguments, namely *CGAL::Tag_false* and *CGAL::Default_algorithm*, respectively.

The template parameters of class *Min_sphere_of_points_d_traits_3<K,FT,UseSqrt,Algorithm>* must fulfill the following requirements:

Min_sphere_of_spheres_d_traits_3<K,FT,UseSqrt,Algorithm>:: K

is a model for *Kernel*.

Min_sphere_of_spheres_d_traits_3<K,FT,UseSqrt,Algorithm>:: FT

is a number type, which fulfills the requirements of type *FT* of concept *MinSphereOfSpheresTraits*: It must be either *double* or *float*, or an exact number type.

Min_sphere_of_spheres_d_traits_3<K,FT,UseSqrt,Algorithm>:: UseSqrt

fulfills the requirements of type *Use_square_roots* of concept *MinSphereOfSpheresTraits*: It must be either *Tag_true* or *Tag_false*.

Min_sphere_of_spheres_d_traits_3<K,FT,UseSqrt,Algorithm>:: Algorithm

fulfills the requirements of type *Algorithm* of concept *MinSphereOfSpheresTraits*: It must be either *Default_algorithm*, *LP_algorithm* or *Farthest_first_heuristic*.

Constants

Min_sphere_of_spheres_d_traits_3<K,FT,UseSqrt,Algorithm>:: D

is the constant 2, i.e. the dimension of $\mathbb{R}^2$.

Types

In addition to the types required by the concept *MinSphereOfSpheresTraits*, this model also defines the types *Radius* and *Point*. Here's the complete list of defined types:

Min_sphere_of_spheres_d_traits_3<K,FT,UseSqrt,Algorithm>:: FT

Min_sphere_of_spheres_d_traits_3<K,FT,UseSqrt,Algorithm>:: Use_square_roots

Min_sphere_of_spheres_d_traits_3<K,FT,UseSqrt,Algorithm>:: Algorithm

Min_sphere_of_spheres_d_traits_3<K,FT,UseSqrt,Algorithm>:: Radius

is a typedef to the template parameter *FT*

Min_sphere_of_spheres_d_traits_3<K,FT,UseSqrt,Algorithm>:: Point

is a typedef to *K::Point_3*.

Min_sphere_of_spheres_d_traits_3<K,FT,UseSqrt,Algorithm>:: Sphere

is a typedef to *Point*.

Min_sphere_of_spheres_d_traits_3<K,FT,UseSqrt,Algorithm>:: Cartesian_const_iterator

is a typedef to *K::Cartesian_const_iterator_3*.

Access Functions

The class provides the access functions required by the concept *MinSphereOfSpheresTraits*; they simply map to the corresponding routines of class *K::Point_3*:

FT *traits.radius(Sphere s)*

returns *0*.

Cartesian_const_iterator

traits.center_cartesian_begin(Sphere s)

maps to *s.first.cartesian_begin()*.

CGAL::Min_sphere_of_points_d_traits_d<K,FT,Dim,UseSqrt,Algorithm>

Definition

The class *Min_sphere_of_points_d_traits_d<K,FT,Dim,UseSqrt,Algorithm>* is a model for concept *MinSphereOfSpheresTraits*. It uses the CGAL type *Point_d* to represent circles.

Is Model for the Concepts

MinSphereOfSpheresTraits

Parameters

The last two template parameters, *UseSqrt* and *Algorithm*, have default arguments, namely *CGAL::Tag_false* and *CGAL::Default_algorithm*, respectively.

The template parameters of class *Min_sphere_of_points_d_traits_d<K,FT,UseSqrt,Algorithm>* must fulfill the following requirements:

Min_sphere_of_points_d_traits_d<K,FT,Dim,UseSqrt,Algorithm>:: K

is a model for *Kernel*.

Min_sphere_of_points_d_traits_d<K,FT,Dim,UseSqrt,Algorithm>:: FT

is a number type, which fulfills the requirements of type *FT* of concept *MinSphereOfSpheresTraits*: It must be either *double* or *float*, or an exact number type.

Min_sphere_of_points_d_traits_d<K,FT,Dim,UseSqrt,Algorithm>:: UseSqrt

fulfills the requirements of type *Use_square_roots* of concept *MinSphereOfSpheresTraits*: It must be either *Tag_true* or *Tag_false*.

Min_sphere_of_points_d_traits_d<K,FT,Dim,UseSqrt,Algorithm>:: Algorithm

fulfills the requirements of type *Algorithm* of concept *MinSphereOfSpheresTraits*: It must be either *Default_algorithm*, *LP_algorithm* or *Farthest_first_heuristic*.

Constants

Min_sphere_of_points_d_traits_d<K,FT,Dim,UseSqrt,Algorithm>:: D

is the constant *Dim*.

Types

In addition to the types required by the concept *MinSphereOfSpheresTraits*, this model also defines the types *Radius* and *Point*. Here's the complete list of defined types:

Min_sphere_of_points_d_traits_d<K,FT,Dim,UseSqrt,Algorithm>:: FT

Min_sphere_of_points_d_traits_d<K,FT,Dim,UseSqrt,Algorithm>:: Use_square_roots

Min_sphere_of_points_d_traits_d<K,FT,Dim,UseSqrt,Algorithm>:: Algorithm

Min_sphere_of_points_d_traits_d<K,FT,Dim,UseSqrt,Algorithm>:: Radius

is a typedef to the template parameter *FT*

Min_sphere_of_points_d_traits_d<K,FT,Dim,UseSqrt,Algorithm>:: Point

is a typedef to *K::Point_d*.

Min_sphere_of_points_d_traits_d<K,FT,Dim,UseSqrt,Algorithm>:: Sphere

is a typedef to *Point*.

Min_sphere_of_points_d_traits_d<K,FT,Dim,UseSqrt,Algorithm>:: Cartesian_const_iterator

is a typedef to *K::Cartesian_const_iterator_d*.

Access Functions

The class provides the access functions required by the concept *MinSphereOfSpheresTraits*; they simply map to the corresponding routines of class *K::Point_d*:

FT *traits.radius(Sphere s)*

returns *0*.

Cartesian_const_iterator

traits.center_cartesian_begin(Sphere s)

maps to *s.cartesian_begin()*.

CGAL::Approximate_min_ellipsoid_d<Traits>

Class

Definition

An object of class *Approximate_min_ellipsoid_d<Traits>* is an approximation to the ellipsoid of smallest volume enclosing a finite multiset of points in d-dimensional Euclidean space $\mathbb{E}^d$, $d \geq 2$.

An *ellipsoid* in $\mathbb{E}^d$ is a Cartesian pointset of the form $\{x \in \mathbb{E}^d \mid x^T E x + x^T e + \eta \leq 0\}$, where E is some positive definite matrix from the set $\mathbb{R}^{d \times d}$, e is some real d-vector, and $\eta \in \mathbb{R}$. A pointset $P \subseteq \mathbb{E}^d$ is called *full-dimensional* if its affine hull has dimension d. For a finite, full-dimensional pointset P we denote by $\text{MEL}(P)$ the smallest ellipsoid that contains all points of P; this ellipsoid exists and is unique.

For a given finite and full-dimensional pointset $P \subset \mathbb{E}^d$ and a real number $\varepsilon \geq 0$, we say that an ellipsoid $\mathcal{E} \subset \mathbb{E}^d$ is an $(1+\varepsilon)$-*appoximation* to $\text{MEL}(P)$ if $P \subset \mathcal{E}$ and $\text{VOL}(\mathcal{E}) \leq (1+\varepsilon)\,\text{VOL}(\text{MEL}(P))$. In other words, an $(1+\varepsilon)$-approximation to $\text{MEL}(P)$ is an enclosing ellipsoid whose volume is by at most a factor of $1+\varepsilon$ larger than the volume of the smallest enclosing ellipsoid of P.

Given this notation, an object of class *Approximate_min_ellipsoid_d<Traits>* represents an $(1+\varepsilon)$-approximation to $\text{MEL}(P)$ for a given finite and full-dimensional multiset of points $P \subset \mathbb{E}^d$ and a real constant $\varepsilon > 0$.[1] When an *Approximate_min_ellipsoid_d<Traits>* object is constructed, an iterator over the points P and the number ε have to be specified; the number ε defines the *desired approximation ratio* $1+\varepsilon$. The underlying algorithm will then try to compute an $(1+\varepsilon)$-approximation to $\text{MEL}(P)$, and one of the following two cases takes place.

- The algorithm determines that P is not full-dimensional (see *is_full_dimensional()* below).

  Important note: due to rounding errors, the algorithm cannot in all cases decide correctly whether P is full-dimensional or not. If *is_full_dimensional()* returns *false*, the points lie in such a "thin" subspace of $\mathbb{E}^d$ that the algorithm is incapable of computing an approximation to $\text{MEL}(P)$. More precisely, if *is_full_dimensional()* returns *false*, there exist two parallel hyperplanes in $\mathbb{E}^d$ with the points P in between so that the distance δ between the hyperplanes is very small, possible zero. (If $\delta = 0$ then P is not full-dimensional.)

  If P is not full-dimensional, linear algebra techniques should be used to determine an affine subspace S of $\mathbb{E}^d$ that contains the points P as a (w.r.t. S) full-dimensional pointset; once S is determined, the algorithm can be invoked again to compute an approximation to (the lower-dimensional) $\text{MEL}(P)$ in S. Since *is_full_dimensional()* might (due to rounding errors, see above) return *false* even though P is full-dimensional, the lower-dimensional subspace S containing P need not exist. Therefore, it might be more advisable to fit a hyperplane H through the pointset P, project P onto this affine subspace H, and compute an approximation to the minimum-volume enclosing ellipsoid of the projected points within H; the fitting can be done for instance using the *linear_least_squares_fitting()* function from the CGAL package *Principal_component_analysis*.

- The algorithm determines that P is full-dimensional. In this case, it provides an approximation $\mathcal{E}$ to $\text{MEL}(P)$, but depending on the input problem (i.e., on the pair (P, ε)), it may not have achieved the desired approximation ratio but merely some *worse* approximation ratio $1+\varepsilon' > 1+\varepsilon$. The achieved approximation ratio $1+\varepsilon'$ can be queried using *achieved_epsilon()*, which returns ε'. The ellipsoid $\mathcal{E}$ itself can be queried via the methods *defining_matrix()*, *defining_vector()*, and *defining_scalar()*.

The ellipsoid $\mathcal{E}$ computed by the algorithm satisfies the inclusions

$$\frac{1}{(1+\varepsilon')d}\mathcal{E} \subseteq \text{conv}(P) \subseteq \mathcal{E} \qquad (66.1)$$

[1] A *multiset* is a set where elements may have multiplicity greater than 1.

where $f\mathcal{E}$ denotes the ellipsoid $\mathcal{E}$ scaled by the factor $f \in \mathbb{R}^+$ with respect to its center, and where $\mathrm{conv}(A)$ denotes the *convex hull* of a pointset $A \subset \mathbb{E}^d$.

The underlying algorithm can cope with all kinds of inputs (multisets P, $\varepsilon \in [0,\infty)$) and terminates in all cases. There is, however, no guarantee that any desired approximation ratio is actually achieved; the performance of the algorithm in this respect highly depends on the input pointset. Values of at least 0.01 for ε are usually handled without problems.

Internally, the algorithm represents the input points' Cartesian coordinates as *double*'s. For this conversion to work, the input point coordinates must be convertible to *double*. Also, in order to compute the achieved epsilon ε' mentioned above, the algorithm requires a number type *ET* that provides *exact* arithmetic. (Both these aspects are discussed in the documentation of the concept *ApproximateMinEllipsoid_d_Traits_d*.)

#include <CGAL/Approximate_min_ellipsoid_d.h>

Requirements

The template parameter *Traits* is a model for *ApproximateMinEllipsoid_d_Traits_d*.

We provide the model *CGAL::Approximate_min_ellipsoid_d_traits_d<K>* using the *d*-dimensional CGAL kernel; the models *CGAL::Approximate_min_ellipsoid_d_traits_2<K>* and *CGAL::Approximate_min_ellipsoid_d_traits_3<K>* are for use with the 2- and 3-dimensional CGAL kernel, respectively.

Types

Approximate_min_ellipsoid_d<Traits>:: FT

typedef Traits::FT FT (which is always a typedef to *double*).

Approximate_min_ellipsoid_d<Traits>:: ET

typedef Traits::ET ET (which is an exact number type used for exact computation like for example in *achieved_epsilon()*).

Approximate_min_ellipsoid_d<Traits>:: Point

typedef Traits::Point Point

Approximate_min_ellipsoid_d<Traits>:: Cartesian_const_iterator

typedef Traits::Cartesian_const_iterator Cartesian_const_iterator

Approximate_min_ellipsoid_d<Traits>:: Center_coordinate_iterator

A model of STL concept *RandomAccessIterator* with value type *double* that is used to iterate over the Cartesian center coordinates of the computed ellipsoid, see *center_cartesian_begin()*.

Approximate_min_ellipsoid_d<Traits>:: Axes_lengths_iterator

A model of STL concept *RandomAccessIterator* with value type *double* that is used to iterate over the lengths of the semiaxes of the computed ellipsoid, see *axes_lengths_begin()*.

Approximate_min_ellipsoid_d<Traits>:: Axis_direction_iterator

A model of STL concept *RandomAccessIterator* with value type *double* that is used to iterate over the Cartesian coordinates of the direction of a fixed axis of the computed ellipsoid, see *axis_direction_cartesian_begin()*.

Creation

An object of type *Approximate_min_ellipsoid_d<Traits>* can be created from an arbitrary point set P and some nonnegative *double* value *eps*.

template < class Iterator >
Approximate_min_ellipsoid_d<Traits> ame(double eps, Iterator first, Iterator last, Traits traits = Traits());

initializes *ame* to an $(1+\varepsilon)$-approximation of $\textsc{mel}(P)$ with P being the set of points in the range [*first*,*last*). The number ε in this will be at most *eps*, if possible. However, due to the limited precision in the algorithm's underlying arithmetic, it can happen that the computed approximation ellipsoid has a worse approximation ratio (and ε can thus be larger than *eps* in general). In any case, the number ε (and with this, the achived approximation $1+\varepsilon$) can be queried by calling the routine *achieved_epsilon()* discussed below.
Requirement: *Iterator* must be a model for concept *InputIterator* with value type *Point*.
Precondition: The dimension d of the input points must be at least 2, and $\varepsilon > 0$.

Access Functions

The following methods can be used to query the achieved approximation ratio $1+\varepsilon'$ and the computed ellipsoid $\mathcal{E} = \{x \in \mathbb{E}^d \mid x^T E x + x^T e + \eta \leq 0\}$. The methods *defining_matrix()*, *defining_vector()*, and *defining_scalar()* do not return E, e, and η directly but yield multiples of these quantities that are exactly representable using the *double* type. (This is necessary because the parameters E, e, and η of the computed approximation ellipsoid $\mathcal{E}$ might not be exactly representable as *double* numbers.)

unsigned int *ame.number_of_points() const*

returns the number of points of *ame*, i.e., $|P|$.

double *ame.achieved_epsilon() const*

returns a number ε' such that the computed approximation is (under exact arithmetic) guaranteed to be an $(1+\varepsilon')$-approximation to $\text{MEL}(P)$.
Precondition: *ame.is_full_dimensional() == true*.
Postcondition: $\varepsilon' > 0$.

double *ame.defining_matrix(int i, int j) const*

gives access to the (i, j)th entry of the matrix E in the representation $\{x \in \mathbb{E}^d \mid x^T E x + x^T e + \eta \leq 0\}$ of the computed approximation ellipsoid $\mathcal{E}$. The number returned by this routine is $(1+\varepsilon')(d+1)\,E_{ij}$, where ε' is the number returned by *achieved_epsilon()*.
Precondition: $0 \leq i, j \leq d$, where d is the dimension of the points P, and *ame.is_full_dimensional() == true*.

double *ame.defining_vector(int i) const*

gives access to the ith entry of the vector e in the representation $\{x \in \mathbb{E}^d \mid x^T E x + x^T e + \eta \leq 0\}$ of the computed approximation ellipsoid $\mathcal{E}$. The number returned by this routine is $(1+\varepsilon')(d+1)\,e_i$, where ε' is the number returned by *achieved_epsilon()*.
Precondition: $0 \leq i \leq d$, where d is the dimension of the points P, and *ame.is_full_dimensional() == true*.

double *ame.defining_scalar() const*

gives access to the scalar η from the representation $\{x \in \mathbb{E}^d \mid x^T E x + x^T e + \eta \leq 0\}$ of the computed approximation ellipsoid $\mathcal{E}$. The number returned by this routine is $(1+\varepsilon')(d+1)\,(\eta+1)$, where ε' is the number returned by *achieved_epsilon()*.
Precondition: *ame.is_full_dimensional() == true*.

Traits *ame.traits() const* returns a const reference to the traits class object.

int *ame.dimension() const*

returns the dimension of the ambient space, i.e., the dimension of the points P.

In order to access the center and semiaxes of the computed approximation ellipsoid, the functions *center_cartesian_begin()*, *axes_lengths_begin()*, and *axis_direction_cartesian_begin()* can be used. In constrast to the above access functions *achieved_epsilon()*, *defining_matrix()*, *defining_vector()*, and *defining_scalar()*, which return the described quantities exactly, the routines below return *numerical approximations* to the real center and real semiaxes of the computed ellipsoid; the comprised relative error may be larger than zero, and there are no guarantees for the returned quantities.

Center_coordinate_iterator
ame.center_cartesian_begin()

returns an iterator pointing to the first of the d Cartesian coordinates of the computed ellipsoid's center.
The returned point is a floating-point approximation to the ellipsoid's exact center; no guarantee is given w.r.t. the involved relative error.
Precondition: *ame.is_full_dimensional() == true*.

Center_coordinate_iterator
ame.center_cartesian_end()

returns the past-the-end iterator corresponding to *center_cartesian_begin()*.
Precondition: *ame.is_full_dimensional() == true*.

Axes_lengths_iterator
ame.axes_lengths_begin()

returns an iterator pointing to the first of the d descendantly sorted lengths of the computed ellipsoid's axes. The d returned numbers are floating-point approximations to the exact axes-lengths of the computed ellipsoid; no guarantee is given w.r.t. the involved relative error. (See also method *axes_direction_cartesian_begin()*.)
Precondition: *ame.is_full_dimensional() == true*, and $d \in \{2,3\}$.

Axes_lengths_iterator
ame.axes_lengths_end()

returns the past-the-end iterator corresponding to *axes_lengths_begin()*.
Precondition: *ame.is_full_dimensional() == true*, and $d \in \{2,3\}$.

Axes_direction_coordinate_iterator
ame.axis_direction_cartesian_begin(int i)

returns an iterator pointing to the first of the d Cartesian coordinates of the computed ellipsoid's ith axis direction (i.e., unit vector in direction of the ellipsoid's ith axis). The direction described by this iterator is a floating-point approximation to the exact axis direction of the computed ellipsoid; no guarantee is given w.r.t. the involved relative error. An approximation to the length of axis i is given by the ith entry of *axes_lengths_begin()*.
Precondition: *ame.is_full_dimensional() == true*, and $d \in \{2,3\}$, and $0 \leq i < d$.

Axes_direction_coordinate_iterator
ame.axis_direction_cartesian_end(int i)

returns the past-the-end iterator corresponding to *axis_direction_cartesian_begin()*.
Precondition: *ame.is_full_dimensional() == true*, and $d \in \{2,3\}$, and $0 \leq i < d$.

Predicates

bool *ame.is_full_dimensional() const*

returns whether P is full-dimensional or not, i.e., returns *true* if and only if P is full-dimensional.
Note: due to the limited precision in the algorithm's underlying arithmetic, the result of this method is not always correct. Rather, a return value of *false* means that the points P are contained in a "very thin" linear subspace of $\mathbb{E}^d$, and as a consequence, the algorithm cannot compute an approximation. More precisely, a return value of *false* means that the points P are contained between two parallel hyperplanes in $\mathbb{E}^d$ that are very close to each other (possibly at distance zero) — so close, that the algorithm could not compute an approximation ellipsoid. Similarly, a return value of *true* does not guarantee P to be full-dimensional; but there exists an input pointset P' such that the points P' and P have almost identical coordinates and P' is full-dimensional.

Validity Check

An object *ame* is valid iff

- *ame* contains all points of its defining set P,
- *ame* is an $(1+\varepsilon')$-approximation to the smallest ellipsoid $\mathrm{MEL}(P)$ of P,
- The ellipsoid represented by *ame* fulfills the inclusion (66.1).

bool *ame.is_valid(bool verbose = false) const*

returns *true* iff *ame* is valid according to the above definition. If *verbose* is *true*, some messages concerning the performed checks are written to the standard error stream.

Miscellaneous

void *ame.write_eps(const std::string& name) const*

Writes the points P and the computed approximation to $\mathrm{MEL}(P)$ as an EPS-file under pathname *name*.
Precondition: The dimension of points P must be 2.
Note: this routine is provided as a debugging routine; future version of CGAL might not provide it anymore.
Precondition: *ame.is_full_dimensional() == true*.

See Also

CGAL::Min_ellipse_2<Traits> . . . . page 3812

Implementation

We implement Khachyian's algorithm for rounding polytopes [Kha96]. Internally, we use *double*-arithmetic and (initially a single) Cholesky-decomposition. The algorithm's running time is $O(nd^2(\varepsilon^{-1}+\ln d+\ln\ln(n)))$, where $n=|P|$ and $1+\varepsilon$ is the desired approximation ratio.

Example

To illustrate the usage of *Approximate_min_ellipsoid_d<Traits>* we give two examples in 2D. The first program generates a random set $P\subset\mathbb{E}^2$ and outputs the points and a 1.01-approximation of $\mathrm{MEL}(P)$ as an EPS-file, which you can view using gv, for instance. (In both examples you can change the variables n and d to experiment with the code.)

```
#include <CGAL/Cartesian_d.h>
#include <CGAL/MP_Float.h>
#include <CGAL/point_generators_d.h>
#include <CGAL/Approximate_min_ellipsoid_d.h>
```

```
#include <CGAL/Approximate_min_ellipsoid_d_traits_d.h>

#include <vector>
#include <iostream>

typedef CGAL::Cartesian_d<double>                              Kernel;
typedef CGAL::MP_Float                                         ET;
typedef CGAL::Approximate_min_ellipsoid_d_traits_d<Kernel, ET> Traits;
typedef Traits::Point                                          Point;
typedef std::vector<Point>
   Point_list;
typedef CGAL::Approximate_min_ellipsoid_d<Traits>              AME;

int main()
{
  const int    n = 1000;                // number of points
  const int    d = 2;                   // dimension
  const double eps = 0.01;              // approximation ratio is
    (1+eps)

  // create a set of random points:
  Point_list P;
  CGAL::Random_points_in_cube_d<Point> rpg(d,100.0);
  for (int i = 0; i < n; ++i) {
    P.push_back(*rpg);
    ++rpg;
  }

  // compute approximation:
  Traits traits;
  AME ame(eps, P.begin(), P.end(), traits);

  // write EPS file:
  if (ame.is_full_dimensional() && d == 2)
    ame.write_eps("example.eps");

  // output center coordinates:
  std::cout << "Cartesian center coordinates: ";
  for (AME::Center_coordinate_iterator c_it =
   ame.center_cartesian_begin();
       c_it != ame.center_cartesian_end();
       ++c_it)
    std::cout << *c_it << ' ';
  std::cout << ".\n";

  if (d == 2 || d == 3) {
    // output  axes:
    AME::Axes_lengths_iterator axes = ame.axes_lengths_begin();
    for (int i = 0; i < d; ++i) {
      std::cout << "Semiaxis " << i << " has length " << *axes++  << "\n"
                << "and Cartesian coordinates ";
      for (AME::Axes_direction_coordinate_iterator
             d_it = ame.axis_direction_cartesian_begin(i);
           d_it != ame.axis_direction_cartesian_end(i); ++d_it)
```

```
        std::cout << *d_it << ' ';
      std::cout << ".\n";
    }
  }
}
```

File: examples/Approximate_min_ellipsoid_d/ellipsoid.cpp

The second program outputs the approximation in a format suitable for display in Maplesoft's Maple.

```
#include <CGAL/Cartesian_d.h>
#include <CGAL/MP_Float.h>
#include <CGAL/point_generators_d.h>
#include <CGAL/Approximate_min_ellipsoid_d.h>
#include <CGAL/Approximate_min_ellipsoid_d_traits_d.h>

#include <vector>
#include <iostream>
#include <iomanip>


typedef CGAL::Cartesian_d<double>                              Kernel;
typedef CGAL::MP_Float                                         ET;
typedef CGAL::Approximate_min_ellipsoid_d_traits_d<Kernel, ET> Traits;
typedef Traits::Point                                          Point;
typedef std::vector<Point>
   Point_list;
typedef CGAL::Approximate_min_ellipsoid_d<Traits>              AME;

int main()
{
  const int       n = 100;                // number of points
  const int       d = 2;                  // dimension
  const double eps = 0.01;                // approximation ratio is
    (1+eps)

  // create a set of random points:
  Point_list P;
  CGAL::Random_points_in_cube_d<Point> rpg(d,1.0);
  for (int i = 0; i < n; ++i) {
    P.push_back(*rpg);
    ++rpg;
  }

  // compute approximation:
  Traits traits;
  AME mel(eps, P.begin(), P.end(), traits);

  // output for Maple:
  if (mel.is_full_dimensional() && d == 2) {

    const double alpha = (1+mel.achieved_epsilon())*(d+1);
```

```
    // output points:
    using std::cout;
    cout << "restart;\n"
         << "with(LinearAlgebra):\n"
         << "with(plottools):\n"
         << "n:= " << n << ":\n"
         << "P:= Matrix(" << d << "," << n << "):\n";

    for (int i=0; i<n; ++i)
      for (int j=0; j<d; ++j)
        cout << "P[" << j+1 << "," << i+1 << "] := "
             << std::setiosflags(std::ios::scientific)
             << std::setprecision(20) << P[i][j] << ":\n";
    cout << "\n";

    // output defining equation:
    cout << "Mp:= Matrix([\n";
    for (int i=0; i<d; ++i) {
      cout << "  [";
      for (int j=0; j<d; ++j) {
        cout << mel.defining_matrix(i,j)/alpha;
        if (j<d-1)
          cout << ",";
      }
      cout << "]";
      if (i<d-1)
        cout << ",";
      cout << "\n";
    }
    cout << "]);\n" << "mp:= Vector([";
    for (int i=0; i<d; ++i) {
      cout << mel.defining_vector(i)/alpha;
      if (i<d-1)
        cout << ",";
    }
    cout << "]);\n"
         << "eta:= " << (mel.defining_scalar()/alpha-1.0) << ";\n"
         << "v:= Vector([x,y]):\n"
         << "e:= Transpose(v).Mp.v+Transpose(v).mp+eta;\n"
         << "plots[display]({seq(point([P[1,i],P[2,i]]),i=1..n),\n"
         << " plots[implicitplot](e,x=-5..5,y=-5..5,numpoints=10000)},\n"
         << " scaling=CONSTRAINED);\n";
  }
}
```

File: examples/Approximate_min_ellipsoid_d/ellipsoid_for_maple.cpp

Concept

ApproximateMinEllipsoid_d_Traits_d

Definition

This concept defines the requirements for traits classes of *CGAL::Approximate_min_ellipsoid_d<Traits>*.

Refines

DefaultConstructible
CopyConstructible
Assignable

Types

ApproximateMinEllipsoid_d_Traits_d:: FT — *typedef double FT*

ApproximateMinEllipsoid_d_Traits_d:: ET — Some model of concept *RingNumberType* that provides exact arithmetic, meaning that *CGAL::Number_type_traits<ET> ::Has_exact_ring_operations* must be *CGAL::Tag_true*. In addition, *ET* must be able to exactly represent any finite *double* value. (An example for such a type is *CGAL::MP_Float*.).
The type *ET* is to be used by the *Approximate_min_ellipsoid_d<Traits>* class for internal, exact computations.

ApproximateMinEllipsoid_d_Traits_d:: Point — Type of the input points. *Point* must provide the default and copy constructor, and must be a model of *DefaultConstructible*, *CopyConstructible*, and *Assignable*.

ApproximateMinEllipsoid_d_Traits_d:: Cartesian_const_iterator

Model for the STL concept *RandomAccessIterator* whose value type must be convertible to *double*. This type is used to iterate over the Cartesian coordinates of an instance of type *Point*, see *cartesian_begin()* below.

Access Functions

int *traits.dimension(Point p)*

returns the dimension of a point *p*.

Cartesian_const_iterator

traits.cartesian_begin(Point p)

returns an input iterator over the Euclidean coordinates of the point *p*. The range of the iterator must have size *dimension(p)*.

Has Models

CGAL::Approximate_min_ellipsoid_d_traits_2<K,ET> . . . . page 3898
CGAL::Approximate_min_ellipsoid_d_traits_3<K,ET> . . . . page 3899
CGAL::Approximate_min_ellipsoid_d_traits_d<K,ET> . . . . page 3900

See Also

CGAL::Min_ellipse_2<Traits> . . . . page 3812

Class

CGAL::Approximate_min_ellipsoid_d_traits_2<K,ET>

Definition

The class *Approximate_min_ellipsoid_d_traits_2<K,ET>* is a traits class for *CGAL::Approximate_min_ellipsoid_d<Traits>* using the 2-dimensional CGAL kernel. In order to use this class, an exact number-type *ET* has to be provided which *Approximate_min_ellipsoid_d<Traits>* will use for its internal exact computations.

#include <CGAL/Approximate_min_ellipsoid_d_traits_2.h>

Requirements

The template parameter *K* must be a model for concept *Kernel*. The template parameter *ET* must be a model for concept *RingNumberType* with exact arithmetic operations, i.e., the type *CGAL::Number_type_traits<ET> ::Has_exact_ring_operations* must be *CGAL::Tag_true*. In addition, *ET* must be able to exactly represent any finite *double* value. (Examples of such a number-type are *CGAL::MP_Float*, *CORE::Expr*, and *CGAL::Gmpq*.)

Is Model for the Concepts

ApproximateMinEllipsoid_d_Traits_d

Types

Approximate_min_ellipsoid_d_traits_2<K,ET>:: FT — *typedef double FT*. The kernel's number type *K::RT* must be convertible to *double*.

Approximate_min_ellipsoid_d_traits_2<K,ET>:: ET — typedef to the second template argument, *ET*.

Approximate_min_ellipsoid_d_traits_2<K,ET>:: Point

typedef K::Point_2 Point

Approximate_min_ellipsoid_d_traits_2<K,ET>:: Cartesian_const_iterator

typedef K::Cartesian_const_iterator_2 Cartesian_const_iterator

See Also

CGAL::Approximate_min_ellipsoid_d_traits_3<K,ET> .... page 3899
CGAL::Approximate_min_ellipsoid_d_traits_d<K,ET> .... page 3900
ApproximateMinEllipsoid_d_Traits_d

CGAL::Approximate_min_ellipsoid_d_traits_3<K,ET>

Class

Definition

The class *Approximate_min_ellipsoid_d_traits_3<K,ET>* is a traits class for *CGAL::Approximate_min_ellipsoid_d<Traits>* using the 3-dimensional CGAL kernel. In order to use this class, an exact number-type *ET* has to be provided which *Approximate_min_ellipsoid_d<Traits>* will use for its internal exact computations.

#include <CGAL/Approximate_min_ellipsoid_d_traits_3.h>

Requirements

The template parameter *K* must be a model for concept *Kernel*. The template parameter *ET* must be a model for concept *RingNumberType* with exact arithmetic operations, i.e., the type *CGAL::Number_type_traits<ET> ::Has_exact_ring_operations* must be *CGAL::Tag_true*. In addition, *ET* must be able to exactly represent any finite *double* value. (Examples of such a number-type are *CGAL::MP_Float*, *CORE::Expr*, and *CGAL::Gmpq*.)

Is Model for the Concepts

ApproximateMinEllipsoid_d_Traits_d

Types

Approximate_min_ellipsoid_d_traits_3<K,ET>:: FT

typedef double FT. The kernel's number type *K::RT* must be convertible to *double*.

Approximate_min_ellipsoid_d_traits_3<K,ET>:: ET

typedef to the second template argument, *ET*.

Approximate_min_ellipsoid_d_traits_3<K,ET>:: Point

typedef K::Point_3 Point

Approximate_min_ellipsoid_d_traits_3<K,ET>:: Cartesian_const_iterator

typedef K::Cartesian_const_iterator_3 Cartesian_const_iterator

See Also

CGAL::Approximate_min_ellipsoid_d_traits_2<K,ET> .... page 3898
CGAL::Approximate_min_ellipsoid_d_traits_d<K,ET> .... page 3900
ApproximateMinEllipsoid_d_Traits_d

Class

CGAL::Approximate_min_ellipsoid_d_traits_d<K,ET>

Definition

The class *Approximate_min_ellipsoid_d_traits_d<K,ET>* is a traits class for *CGAL::Approximate_min_ellipsoid_d<Traits>* using the *d*-dimensional CGAL kernel. In order to use this class, an exact number-type *ET* has to be provided which *Approximate_min_ellipsoid_d<Traits>* will use for its internal exact computations.

#include <CGAL/Approximate_min_ellipsoid_d_traits_d.h>

Requirements

The template parameter *K* must be a model for concept *Kernel*. The template parameter *ET* must be a model for concept *RingNumberType* with exact arithmetic operations, i.e., the type *CGAL::Number_type_traits<ET> ::Has_exact_ring_operations* must be *CGAL::Tag_true*. In addition, *ET* must be able to exactly represent any finite *double* value. (Examples of such a number-type are *CGAL::MP_Float*, *CORE::Expr*, and *CGAL::Gmpq*.)

Is Model for the Concepts

ApproximateMinEllipsoid_d_Traits_d

Types

Approximate_min_ellipsoid_d_traits_d<K,ET>:: FT

typedef double FT. The kernel's number type *K::RT* must be convertible to *double*.

Approximate_min_ellipsoid_d_traits_d<K,ET>:: ET

typedef to the second template argument, *ET*.

Approximate_min_ellipsoid_d_traits_d<K,ET>:: Point

typedef K::Point_d Point

Approximate_min_ellipsoid_d_traits_d<K,ET>:: Cartesian_const_iterator

typedef K::Cartesian_const_iterator Cartesian_const_iterator

See Also

CGAL::Approximate_min_ellipsoid_d_traits_2<K,ET> . . . . page 3898
CGAL::Approximate_min_ellipsoid_d_traits_3<K,ET> . . . . page 3899
ApproximateMinEllipsoid_d_Traits_d

CGAL::Min_sphere_annulus_d_traits_2<K,ET,NT>

Class

Definition

The class *Min_sphere_annulus_d_traits_2<K,ET,NT>* is a traits class for the d-dimensional optimisation algorithms using the two-dimensional CGAL kernel.

#include <CGAL/Min_sphere_annulus_d_traits_2.h>

Requirements

The template parameter *K* is a model for *Kernel*. Template parameters *ET* and *NT* are models for *RingNumberType*.

The second and third template parameter have default type *K::RT*.

Is Model for the Concepts

MinSphereAnnulusDTraits . . . . page 3907

Types

Min_sphere_annulus_d_traits_2<K,ET,NT>:: Point_d	typedef to *K::Point_2*.
Min_sphere_annulus_d_traits_2<K,ET,NT>:: Rep_tag	typedef to *K::Rep_tag*.
Min_sphere_annulus_d_traits_2<K,ET,NT>:: RT	typedef to *K::RT*.
Min_sphere_annulus_d_traits_2<K,ET,NT>:: FT	typedef to *K::FT*.
Min_sphere_annulus_d_traits_2<K,ET,NT>:: Access_dimension_d	typedef to *K::Access_dimension_2*.
Min_sphere_annulus_d_traits_2<K,ET,NT>:: Access_coordinates_begin_d	typedef to *K::Access_coordinates_begin_2*.
Min_sphere_annulus_d_traits_2<K,ET,NT>:: Construct_point_d	typedef to *K::Construct_point_2*.
Min_sphere_annulus_d_traits_2<K,ET,NT>:: ET	second template parameter (default is *K::RT*).
Min_sphere_annulus_d_traits_2<K,ET,NT>:: NT	third template parameter (default is *K::RT*).

Creation

Min_sphere_annulus_d_traits_2<K,ET,NT> traits; default constructor.

Min_sphere_annulus_d_traits_2<K,ET,NT> traits(Min_sphere_annulus_d_traits_2<K,ET,NT>);

copy constructor.

Operations

The following functions just return the corresponding function class object.

Access_dimension_d	*traits.access_dimension_d_object() const*
Access_coordinates_begin_d	*traits.access_coordinates_begin_d_object() const*
Construct_point_d	*traits.construct_point_d_object() const*

See Also

CGAL::Min_sphere_d<Traits> . . . . page 3846
CGAL::Min_annulus_d<Traits> . . . . page 3860
CGAL::Min_sphere_annulus_d_traits_3<K,ET,NT> . . . . page 3903
CGAL::Min_sphere_annulus_d_traits_d<K,ET,NT> . . . . page 3905

MinSphereAnnulusDTraits . . . . page 3907

CGAL::Min_sphere_annulus_d_traits_3<K,ET,NT>

Class

Definition

The class *Min_sphere_annulus_d_traits_3<K,ET,NT>* is a traits class for the d-dimensional optimisation algorithms using the three-dimensional CGAL kernel.

#include <CGAL/Min_sphere_annulus_d_traits_3.h>

Requirements

The template parameter *K* is a model for *Kernel*. Template parameters *ET* and *NT* are models for *RingNumberType*.

The second and third template parameter have default type *K::RT*.

Is Model for the Concepts

MinSphereAnnulusDTraits . . . . page 3907

Types

Min_sphere_annulus_d_traits_3<K,ET,NT>:: Point_d — typedef to *K::Point_3*.

Min_sphere_annulus_d_traits_3<K,ET,NT>:: Rep_tag — typedef to *K::Rep_tag*.

Min_sphere_annulus_d_traits_3<K,ET,NT>:: RT — typedef to *K::RT*.

Min_sphere_annulus_d_traits_3<K,ET,NT>:: FT — typedef to *K::FT*.

Min_sphere_annulus_d_traits_3<K,ET,NT>:: Access_dimension_d

typedef to *K::Access_dimension_3*.

Min_sphere_annulus_d_traits_3<K,ET,NT>:: Access_coordinates_begin_d

typedef to *K::Access_coordinates_begin_3*.

Min_sphere_annulus_d_traits_3<K,ET,NT>:: Construct_point_d

typedef to *K::Construct_point_3*.

Min_sphere_annulus_d_traits_3<K,ET,NT>:: ET — second template parameter (default is *K::RT*).

Min_sphere_annulus_d_traits_3<K,ET,NT>:: NT — third template parameter (default is *K::RT*).

Creation

Min_sphere_annulus_d_traits_3<K,ET,NT> traits; default constructor.

Min_sphere_annulus_d_traits_3<K,ET,NT> traits(Min_sphere_annulus_d_traits_3<K,ET,NT>);

copy constructor.

Operations

The following functions just return the corresponding function class object.

Access_dimension_d *traits.access_dimension_d_object() const*

Access_coordinates_begin_d *traits.access_coordinates_begin_d_object() const*

Construct_point_d *traits.construct_point_d_object() const*

See Also

CGAL::Min_sphere_d<Traits> . . . . page 3846
CGAL::Min_annulus_d<Traits> . . . . page 3860
CGAL::Min_sphere_annulus_d_traits_2<K,ET,NT> . . . . page 3901
CGAL::Min_sphere_annulus_d_traits_d<K,ET,NT> . . . . page 3905
Min_sphere_annulusDTraits . . . . page **??**

CGAL::Min_sphere_annulus_d_traits_d<K,ET,NT>

Class

Definition

The class *Min_sphere_annulus_d_traits_d<K,ET,NT>* is a traits class for the d-dimensional optimisation algorithms using the d-dimensional CGAL kernel.

#include <CGAL/Min_sphere_annulus_d_traits_d.h>

Requirements

The template parameter *K* is a model for *Kernel*. Template parameters *ET* and *NT* are models for *RingNumberType*.

The second and third template parameter have default type *K::RT*.

Is Model for the Concepts

MinSphereAnnulusDTraits . . . . page 3907

Types

Min_sphere_annulus_d_traits_d<K,ET,NT>:: Point_d	typedef to *K::Point_d*.
Min_sphere_annulus_d_traits_d<K,ET,NT>:: Rep_tag	typedef to *K::Rep_tag*.
Min_sphere_annulus_d_traits_d<K,ET,NT>:: RT	typedef to *K::RT*.
Min_sphere_annulus_d_traits_d<K,ET,NT>:: FT	typedef to *K::FT*.
Min_sphere_annulus_d_traits_d<K,ET,NT>:: Access_dimension_d	typedef to *K::Access_dimension_d*.
Min_sphere_annulus_d_traits_d<K,ET,NT>:: Access_coordinates_begin_d	typedef to *K::Access_coordinates_begin_d*.
Min_sphere_annulus_d_traits_d<K,ET,NT>:: Construct_point_d	typedef to *K::Construct_point_d*.
Min_sphere_annulus_d_traits_d<K,ET,NT>:: ET	second template parameter (default is *K::RT*).
Min_sphere_annulus_d_traits_d<K,ET,NT>:: NT	third template parameter (default is *K::RT*).

Creation

Min_sphere_annulus_d_traits_d<K,ET,NT> traits; default constructor.

Min_sphere_annulus_d_traits_d<K,ET,NT> traits(Min_sphere_annulus_d_traits_d<K,ET,NT>);

copy constructor.

Operations

The following functions just return the corresponding function class object.

Access_dimension_d	*traits.access_dimension_d_object() const*
Access_coordinates_begin_d	*traits.access_coordinates_begin_d_object() const*
Construct_point_d	*traits.construct_point_d_object() const*

See Also

CGAL::Min_sphere_d<Traits> . . . . page 3846
CGAL::Min_annulus_d<Traits> . . . . page 3860
CGAL::Polytope_distance_d<Traits> . . . . page 3944
CGAL::Min_sphere_annulus_d_traits_2<K,ET,NT> . . . . page 3901
CGAL::Min_sphere_annulus_d_traits_3<K,ET,NT> . . . . page 3903
MinSphereAnnulusDTraits . . . . page 3907

Concept

MinSphereAnnulusDTraits

Definition

This concept defines the requirements for traits classes of d-dimensional min sphere and min annulus algorithms.

Types

MinSphereAnnulusDTraits:: Point_d — point type used to represent the input points.

MinSphereAnnulusDTraits:: Rep_tag — compile time tag to distinguish between Cartesian and homogeneous representation of the input points. *Rep_tag* has to be either *CGAL::Cartesian_tag* or *CGAL::Homogeneous_tag*.

MinSphereAnnulusDTraits:: RT — number type used to represent the coordinates of the input points. It has to be a model for *RingNumberType*.

MinSphereAnnulusDTraits:: FT — number type used to return either the squared radius of the smallest enclosing sphere or annulus, or the squared distance of the polytopes. *FT* has to be either *RT* or *CGAL::Quotient*<*RT*> if the input points have Cartesian or homogeneous representation, respectively (cf. *Rep_tag*).

MinSphereAnnulusDTraits:: Access_dimension_d

data accessor object used to access the dimension of the input points.

MinSphereAnnulusDTraits:: Access_coordinates_begin_d

data accessor object used to access the coordinates of the input points.

MinSphereAnnulusDTraits:: Construct_point_d

constructor object used to construct either the center of the smallest enclosing sphere or annulus, or the points realizing the distance between the two polytopes.

The following two number types are only needed for *CGAL::Min_annulus_d*<*Traits*> and *CGAL::Polytope_distance_d*<*Traits*>.

MinSphereAnnulusDTraits:: ET — exact number type used to do the exact computations in the underlying solver for linear programs. It has to to be a model for *RingNumberType*. There must be an implicit conversion from *RT* to *ET* available.

MinSphereAnnulusDTraits:: NT — fast (possibly inexact) number type used to speed up the pricing step in the underlying solver for linear programs. It has to be a model for *RingNumberType*. There must be implicit conversions from *RT* to *NT* and from *NT* to *ET* available.

Creation

Only default and copy constructor are required.

MinSphereAnnulusDTraits traits;

MinSphereAnnulusDTraits traits(MinSphereAnnulusDTraits);

Operations

The following functions just return the corresponding function class object.

Access_dimension_d	*traits.access_dimension_d_object() const*
Access_coordinates_begin_d	*traits.access_coordinates_begin_d_object() const*
Construct_point_d	*traits.construct_point_d_object() const*

Has Models

CGAL::Min_sphere_annulus_d_traits_2<K,ET,NT> . . . . page 3901
CGAL::Min_sphere_annulus_d_traits_3<K,ET,NT> . . . . page 3903
CGAL::Min_sphere_annulus_d_traits_d<K,ET,NT> . . . . page 3905

See Also

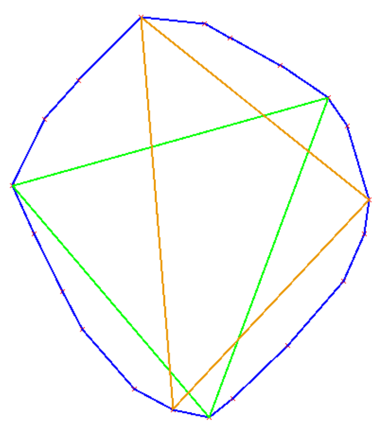
CGAL::Min_sphere_d<Traits> . . . . page 3846
CGAL::Min_annulus_d<Traits> . . . . page 3860

Chapter 67

Inscribed Areas

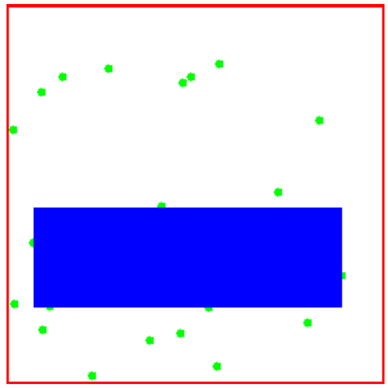
Michael Hoffmann and Eli Packer

This chapter describes algorithms which for a given point set compute the "best" inscribed object from a specific class. We provide algorithms for computing maximal inscribed k-gons (triangles, quadrilaterals, ...) of a planar point set P. Maximal k-gons are convex, and it is known that their vertices can be chosen to be vertices of the convex hull of P. Hence, the functions *CGAL::maximum_area_inscribed_k_gon_2* and *CGAL::maximum_perimeter_inscribed_k_gon_2* operate on convex polygons only. The example below shows that the largest area triangle (green) and the largest perimeter triangle (orange, containing the top point) of a point set are different in general.



We further provide an algorithm for computing the maximal area inscribed axis parallel rectangle

Given a set of points in the plane, the class *CGAL::Largest_empty_iso_rectangle_2<T>* is a data structure that maintains an iso-rectangle with the largest area among all iso-rectangles that are inside a given iso-rectangles, and that do not contain any point of the point set.



Inscribed volumes are also frequently applied to extract geometric properties of objects. The largest area triangle is for example used in heuristics for matching archaeological aerial photographs. Largest perimeter triangles are used in scoring cross country soaring flights, where the goal is basically to fly as far as possible, but still return to the departure airfield. To score simply based on the total distance flown is not a good measure, since circling in thermals allows to increase it easily.

Inscribed Areas
Reference Manual

Michael Hoffmann and Eli Packer

This chapter describes concepts, classes, and functions for maximum area and perimeter inscribed k-gon (2D), extremal inscribed k-gon (2D), largest empty isorectangle (2D).

Assertions

The optimization code uses infix *OPTIMISATION* in the assertions, e.g. defining the compiler flag *CGAL_OPTIMISATION_NO_PRECONDITIONS* switches precondition checking off, cf. Section 2.8.

67.1 Classified References Pages

67.2 Alphabetical List of Reference Pages

Function

CGAL::maximum_area_inscribed_k_gon_2

Definition

The function *maximum_area_inscribed_k_gon_2* computes a maximum area k-gon P_k that can be inscribed into a given convex polygon P. Note that

- P_k is not unique in general, but it can be chosen in such a way that its vertices form a subset of the vertex set of P and
- the vertices of a maximum area k-gon, where the k vertices are to be drawn from a planar point set S, lie on the convex hull of S i.e. a convex polygon.

#include <CGAL/extremal_polygon_2.h>

```
template < class RandomAccessIterator, class OutputIterator >
OutputIterator        maximum_area_inscribed_k_gon_2(
                          RandomAccessIterator points_begin,
                          RandomAccessIterator points_end,
                          int k,
                          OutputIterator o)
```

computes a maximum area inscribed k-gon of the convex polygon described by [*points_begin*, *points_end*), writes its vertices to o and returns the past-the-end iterator of this sequence.

Precondition:

1. the – at least three – points denoted by the range [*points_begin*, *points_end*) form the boundary of a convex polygon (oriented clock– or counterclockwise).
2. $k \geq 3$.

Requirement:

1. Value type of *RandomAccessIterator* is *K::Point_2* where K is a model for *Kernel*.
2. *OutputIterator* accepts the value type of *RandomAccessIterator* as value type.

See Also

CGAL::maximum_perimeter_inscribed_k_gon_2 . . . . page 3914
ExtremalPolygonTraits_2 . . . . page 3921
CGAL::Extremal_polygon_area_traits_2<K> . . . . page 3917
CGAL::Extremal_polygon_perimeter_traits_2<K> . . . . page 3919
CGAL::extremal_polygon_2 . . . . page 3916
CGAL::monotone_matrix_search . . . . page 458

Implementation

The implementation uses monotone matrix search [AKM+87] and has a worst case running time of $O(k \cdot n + n \cdot \log n)$, where n is the number of vertices in P.

Example

The following code generates a random convex polygon p with ten vertices and computes the maximum area inscribed five-gon of p.

```
#include <CGAL/Cartesian.h>
#include <CGAL/Polygon_2.h>
#include <CGAL/point_generators_2.h>
#include <CGAL/random_convex_set_2.h>
#include <CGAL/extremal_polygon_2.h>
#include <iostream>
#include <vector>

typedef double                                         FT;

typedef CGAL::Cartesian<FT>                            Kernel;

typedef Kernel::Point_2                                Point;
typedef std::vector<int>                               Index_cont;
typedef CGAL::Polygon_2<Kernel>                        Polygon_2;
typedef CGAL::Random_points_in_square_2<Point>         Generator;

int main() {

  int n = 10;
  int k = 5;

  // generate random convex polygon:
  Polygon_2 p;
  CGAL::random_convex_set_2(n, std::back_inserter(p), Generator(1));
  std::cout << "Generated Polygon:\n" << p << std::endl;

  // compute maximum area incribed k-gon of p:
  Polygon_2 k_gon;
  CGAL::maximum_area_inscribed_k_gon_2(
    p.vertices_begin(), p.vertices_end(), k, std::back_inserter(k_gon));
  std::cout << "Maximum area " << k << "-gon:\n"
            << k_gon << std::endl;

  return 0;
}
```

File: examples/Matrix_search/extremal_polygon_2_area.cpp

Function

CGAL::maximum_perimeter_inscribed_k_gon_2

Definition

The function *maximum_perimeter_inscribed_k_gon_2* computes a maximum perimeter k-gon P_k that can be inscribed into a given convex polygon P. Note that

- P_k is not unique in general, but it can be chosen in such a way that its vertices form a subset of the vertex set of P and
- the vertices of a maximum perimeter k-gon, where the k vertices are to be drawn from a planar point set S, lie on the convex hull of S i.e. a convex polygon.

#include <CGAL/extremal_polygon_2.h>

```
template < class RandomAccessIterator, class OutputIterator >
OutputIterator      maximum_perimeter_inscribed_k_gon_2(
                           RandomAccessIterator points_begin,
                           RandomAccessIterator points_end,
                           int k,
                           OutputIterator o)
```

computes a maximum perimeter inscribed k-gon of the convex polygon described by [*points_begin*, *points_end*), writes its vertices to o and returns the past-the-end iterator of this sequence.

Precondition:

1. the – at least three – points denoted by the range [*points_begin*, *points_end*) form the boundary of a convex polygon (oriented clock– or counterclockwise).
2. $k \geq 2$.

Requirement:

1. Value type of *RandomAccessIterator* is *K::Point_2* where K is a model for *Kernel*.
2. There is a global function *K::FT CGAL::sqrt(K::FT)* defined that computes the squareroot of a number.
3. *OutputIterator* accepts the value type of *RandomAccessIterator* as value type.

See Also

CGAL::maximum_area_inscribed_k_gon_2 . . . . . page 3912
ExtremalPolygonTraits_2 . . . . . page 3921
CGAL::Extremal_polygon_area_traits_2<K> . . . . . page 3917
CGAL::Extremal_polygon_perimeter_traits_2<K> . . . . . page 3919
CGAL::extremal_polygon_2 . . . . . page 3916
CGAL::monotone_matrix_search . . . . . page 458

Implementation

The implementation uses monotone matrix search [AKM+87] and has a worst case running time of $O(k \cdot n + n \cdot \log n)$, where n is the number of vertices in P.

Example

The following code generates a random convex polygon p with ten vertices and computes the maximum perimeter inscribed five-gon of p.

```
#include <CGAL/Cartesian.h>
#include <CGAL/Polygon_2.h>
#include <CGAL/point_generators_2.h>
#include <CGAL/random_convex_set_2.h>
#include <CGAL/extremal_polygon_2.h>
#include <iostream>
#include <vector>

typedef double                                    FT;

typedef CGAL::Cartesian<FT>                       Kernel;

typedef Kernel::Point_2                           Point;
typedef std::vector<int>                          Index_cont;
typedef CGAL::Polygon_2<Kernel>                   Polygon_2;
typedef CGAL::Random_points_in_square_2<Point>    Generator;

int main() {

  int n = 10;
  int k = 5;

  // generate random convex polygon:
  Polygon_2 p;
  CGAL::random_convex_set_2(n, std::back_inserter(p), Generator(1));
  std::cout << "Generated Polygon:\n" << p << std::endl;

  // compute maximum perimeter incribed k-gon of p:
  Polygon_2 k_gon;
  CGAL::maximum_perimeter_inscribed_k_gon_2(
    p.vertices_begin(), p.vertices_end(), k, std::back_inserter(k_gon));
  std::cout << "Maximum perimeter " << k << "-gon:\n"
            << k_gon << std::endl;

  return 0;
}
```

File: examples/Matrix_search/extremal_polygon_2_perimeter.cpp

Function

CGAL::extremal_polygon_2

advanced

Definition

The function *extremal_polygon_2* computes a maximal k-gon that can be inscribed into a given convex polygon. The criterion for maximality and some basic operations have to be specified in an appropriate traits class parameter.

#include <CGAL/extremal_polygon_2.h>

```
template < class RandomAccessIterator, class OutputIterator, class Traits >
OutputIterator      extremal_polygon_2(
                        RandomAccessIterator points_begin,
                        RandomAccessIterator points_end,
                        int k,
                        OutputIterator o,
                        Traits t)
```

computes a maximal (as specified by t) inscribed k-gon of the convex polygon described by [*points_begin*, *points_end*), writes its vertices to o and returns the past-the-end iterator of this sequence.

Precondition:

1. the – at least three – points denoted by the range [*points_begin*, *points_end*) form the boundary of a convex polygon (oriented clock– or counterclockwise).
2. $k \geq t.min\_k()$.

Requirement:

1. *Traits* is a model for *ExtremalPolygonTraits_2*.
2. Value type of *RandomAccessIterator* is *Traits::Point_2*.
3. *OutputIterator* accepts *Traits::Point_2* as value type.

See Also

CGAL::maximum_area_inscribed_k_gon_2 . . . . page 3912
CGAL::maximum_perimeter_inscribed_k_gon_2 . . . . page 3914
ExtremalPolygonTraits_2 . . . . page 3921
CGAL::monotone_matrix_search . . . . page 458

Implementation

The implementation uses monotone matrix search [AKM+87] and has a worst case running time of $O(k \cdot n + n \cdot \log n)$, where n is the number of vertices in P.

advanced

CGAL::Extremal_polygon_area_traits_2<K>

advanced

Definition

The class provides the types and operations needed to compute a maximum area k-gon P_k that can be inscribed into a given convex polygon P using the function *extremal_polygon_2*.

Requirements

The template parameter K is a model for *Kernel*.

Is Model for the Concepts

ExtremalPolygonTraits_2 . . . . page 3921

Types

Extremal_polygon_area_traits_2<K>:: FT — typedef to *K::FT*.

Extremal_polygon_area_traits_2<K>:: Point_2 — typedef to *K::Point_2*.

Extremal_polygon_area_traits_2<K>:: Less_xy_2

typedef to *K::Less_xy_2*.

Extremal_polygon_area_traits_2<K>:: Orientation_2

typedef to *K::Orientation_2*.

Extremal_polygon_area_traits_2<K>:: Operation

AdaptableBinaryFunction class op: $Point\_2 \times Point\_2 \rightarrow FT$. For a fixed *Point_2 root*, $op(p,q)$ returns twice the area of the triangle $(root, q, p)$.

Operations

int	*t.min_k() const*	returns 3.
FT	*t.init(const Point_2& p, const Point_2& q) const*	returns *FT(0)*.

Class

Operation *t.operation(const Point_2& p) const*

returns *Operation* where *p* is the fixed *root* point.

template < class RandomAccessIterator, class OutputIterator >
OutputIterator *t.compute_min_k_gon(RandomAccessIterator points_begin, RandomAccessIterator points_end, FT& max_area, OutputIterator o) const*

writes the vertices of [*points_begin*, *points_end*) forming a maximum area triangle rooted at *points_begin[0]* to o and returns the past-the-end iterator for that sequence (== *o* + *3*).

Less_xy_2 *t.less_xy_2_object()*
Orientation_2 *t.orientation_2_object()*

See Also

CGAL::maximum_area_inscribed_k_gon_2 . . . . page 3912
CGAL::maximum_perimeter_inscribed_k_gon_2 . . . . page 3914
CGAL::extremal_polygon_2 . . . . page 3916
CGAL::Extremal_polygon_perimeter_traits_2<K> . . . . page 3919
ExtremalPolygonTraits_2 . . . . page 3921

advanced

CGAL::Extremal_polygon_perimeter_traits_2<K>

advanced

Definition

The class provides the types and operations needed to compute a maximum perimeter k-gon P_k that can be inscribed into a given convex polygon P using the function *extremal_polygon_2*.

Requirements

The template parameter K is a model for *Kernel*.

Is Model for the Concepts

ExtremalPolygonTraits_2 . . . . page 3921

Types

Extremal_polygon_perimeter_traits_2<K>:: FT

typedef to *K::FT*.

Extremal_polygon_perimeter_traits_2<K>:: Point_2

typedef to *K::Point_2*.

Extremal_polygon_perimeter_traits_2<K>:: Less_xy_2

typedef to *K::Less_xy_2*.

Extremal_polygon_perimeter_traits_2<K>:: Orientation_2

typedef to *K::Orientation_2*.

Extremal_polygon_perimeter_traits_2<K>:: Operation

AdaptableBinaryFunction class *op*: *Point_2* $\times$ *Point_2* $\rightarrow$ *FT*. For a fixed *Point_2 root*, $op(p,q)$ returns $d(r,p) + d(p,q) - d(r,q)$ where d denotes the Euclidean distance.

Operations

int	*t.min_k() const*	returns 2.
FT	*t.init(const Point_2& p, const Point_2& q) const*	returns twice the Euclidean distance between *p* and *q*.
Operation	*t.operation(const Point_2& p) const*	returns *Operation* where *p* is the fixed *root* point.

template < class RandomAccessIterator, class OutputIterator >
OutputIterator *t.compute_min_k_gon(RandomAccessIterator points_begin, RandomAccessIterator points_end, FT& max_area, OutputIterator o) const*

writes the pair (*points_begin[0]*, *p*) where *p* is drawn from [*points_begin*, *points_end*) such that the Euclidean distance between both points is maximized (maximum perimeter 2-gon rooted at *points_begin[0]*) to o and returns the past-the-end iterator for that sequence (== *o + 2*).

Less_xy_2	*t.less_xy_2_object()*
Orientation_2	*t.orientation_2_object()*

See Also

CGAL::maximum_area_inscribed_k_gon_2 . . . . page 3912
CGAL::maximum_perimeter_inscribed_k_gon_2 . . . . page 3914
CGAL::extremal_polygon_2 . . . . page 3916
CGAL::Extremal_polygon_area_traits_2<K> . . . . page 3917
ExtremalPolygonTraits_2 . . . . page 3921

advanced

ExtremalPolygonTraits_2

advanced

Definition

The concept ExtremalPolygonTraits_2 provides the types and operations needed to compute a maximal k-gon that can be inscribed into a given convex polygon.

Types

ExtremalPolygonTraits_2:: FT	model for FieldNumberType . . . . page 149.
ExtremalPolygonTraits_2:: Point_2	model for Kernel::Point_2 . . . . page 794.
ExtremalPolygonTraits_2:: Less_xy_2	model for Kernel::Less_xy_2 . . . . page **??**.
ExtremalPolygonTraits_2:: Orientation_2	model for Kernel::Orientation_2 . . . . page 1034.
ExtremalPolygonTraits_2:: Operation	AdaptableBinaryFunction class *op*: $Point\_2 \times Point\_2 \rightarrow FT$. Together with *init* this operation recursively defines the objective function to maximize. Let p and q be two vertices of a polygon P such that q precedes p in the oriented vertex chain of P starting with vertex $root$. Then *op(p,q)* returns the value by which an arbitrary sub-polygon of P with vertices from $[root, q]$ increases when p is added to it. E.g. in the maximum area case this is the area of the triangle $(root, q, p)$.

Operations

int *t.min_k() const*

returns the minimal k for which a maximal k-gon can be computed. (e.g. in the maximum area case this is three.)

FT *t.init(const Point_2& p, const Point_2& q) const*

returns the value of the objective function for a polygon consisting of the two points p and q. (e.g. in the maximum area case this is *FT(0)*.)

Operation *t.operation(const Point_2& p) const*

return *Operation* where p is the fixed *root* point.

template < class RandomAccessIterator, class OutputIterator >

Concept

OutputIterator	*t.compute_min_k_gon(*	*RandomAccessIterator points_begin,* *RandomAccessIterator points_end,* *FT& max_area,* *OutputIterator o) const*

writes the points of [*points_begin*, *points_end*) forming a *min_k()*-gon rooted at *points_begin[0]* of maximal value to o and returns the past-the-end iterator for that sequence (== *o* + *min_k()*).

Less_xy_2	*t.less_xy_2_object()*
Orientation_2	*t.orientation_2_object()*

Notes

- *::Less_xy_2* and *::Orientation_2* are used for (expensive) precondition checking only. Therefore, they need not to be specified, in case that precondition checking is disabled.

Has Models

CGAL::Extremal_polygon_area_traits_2<K> .... page 3917
CGAL::Extremal_polygon_perimeter_traits_2<K> .... page 3919

See Also

CGAL::maximum_area_inscribed_k_gon_2 .... page 3912
CGAL::maximum_perimeter_inscribed_k_gon_2 .... page 3914
CGAL::extremal_polygon_2 .... page 3916

advanced

CGAL::Largest_empty_iso_rectangle_2<T>

Class

Definition

Given a set of points in the plane, the class *Largest_empty_iso_rectangle_2<T>* is a data structure that maintains an iso-rectangle with the largest area among all iso-rectangles that are inside a given bounding box(iso-rectangle), and that do not contain any point of the point set.

The class *Largest_empty_iso_rectangle_2<T>* expects a model of the concept *LargestEmptyIsoRectangleTraits_2* as its template argument.

#include <CGAL/Largest_empty_iso_rectangle_2.h>

Types

The class *Largest_empty_iso_rectangle_2<T>* defines the following types:

typedef T *Traits;*

typedef Traits::Point_2

Point_2;

typedef Traits::Iso_rectangle_2

Iso_rectangle_2;

The following iterator allows to enumerate the points. It is non mutable, bidirectional and its value type is *Point_2*. It is invalidated by any insertion or removal of a point.

Largest_empty_iso_rectangle_2<T>:: const_iterator

Iterator over the points.

Creation

Largest_empty_iso_rectangle_2<T> l(Iso_rectangle_2 b);

Constructor. The iso-rectangle b is the bounding rectangle.

Largest_empty_iso_rectangle_2<T> l(const Point_2 p, const Point_2 q);

Constructor. The iso-rectangle whose lower left and upper right points are p and q respectively is the bounding rectangle.

Largest_empty_iso_rectangle_2<T> l;

Constructor. The iso-rectangle whose lower left point and upper right points are (0,0) and (1,1) respectively is the bounding rectangle.

Largest_empty_iso_rectangle_2<T> l(const Largest_empty_iso_rectangle_2<Traits> tr);

Copy constructor.

Operations

Assignment

Largest_empty_iso_rectangle_2<T>

l = tr

Access Functions

Traits	*l.traits() const*	Returns a const reference to the traits object.
const_iterator	*l.begin() const*	Returns an iterator to the beginning of the point set.
const_iterator	*l.end() const*	Returns a past-the-end iterator for the point set.

Queries

Quadruple<Point_2, Point_2, Point_2, Point_2>

l.get_left_bottom_right_top()

Returns the four points that define the largest empty iso-rectangle. (Note that these points are not necessarily on a corner of an iso-rectangle.)

Iso_rectangle_2

l.get_largest_empty_iso_rectangle()

Returns the largest empty iso-rectangle. (Note that the two points defining the iso-rectangle are not necessarily part of the point set.)

Iso_rectangle_2

l.get_bounding_box() Returns the iso-rectangle passed in the constructor.

Insertion

void	*l.insert(Point_2 p)*	Inserts point p in the point set, if it is not already in the set.
void	*l.push_back(Point_2 p)*	Inserts point p in the point set, if it is not already in the set.

template < class InputIterator >

int *l.insert(InputIterator first, InputIterator last)*

Inserts the points in the range [*first*, *last*). Returns the number of inserted points.

Requirements

The *value_type* of *first* and *last* is *Point*.

Removal

bool	*l.remove(Point_2 p)*	Removes point p. Returns false iff p is not in the point set.
void	*l.clear()*	Removes all points of l.

Implementation

The algorithm is an implementation of [Orl90]. The runtime of an insertion or a removal is $O(\log n)$. A query takes $O(n^2)$ worst case time and $O(n \log n)$ expected time. The working storage is $O(n)$.

Concept

LargestEmptyIsoRectangleTraits_2

Definition

The concept LargestEmptyIsoRectangleTraits_2 describes the set of requirements to be fulfilled by any class used to instantiate the template parameter of the class *Largest_empty_iso_rectangle_2<T>*. This concept provides the types of the geometric primitives used in this class and some function object types for the required predicates on those primitives.

Types

LargestEmptyIsoRectangleTraits_2:: Point_2 — The point type.

LargestEmptyIsoRectangleTraits_2:: Iso_rectangle_2

The iso rectangle type.

LargestEmptyIsoRectangleTraits_2:: Compare_x_2

Predicate object. Must provide the operator *Comparison_result operator()(Point_2 p, Point_2 q)* which returns *SMALLER, EQUAL* or *LARGER* according ding to the x-ordering of points p and q.

LargestEmptyIsoRectangleTraits_2:: Compare_y_2

Predicate object. Must provide the operator *Comparison_result operator()(Point_2 p, Point_2 q)* which returns *SMALLER, EQUAL* or *LARGER* according to the y-ordering of points p and q.

LargestEmptyIsoRectangleTraits_2:: Less_x_2 — Predicate object. Must provide the operator *bool operator()(Point_2 p, Point_2 q)* which returns whether p is less than q according to their x-ordering.

LargestEmptyIsoRectangleTraits_2:: Less_y_2 — Predicate object. Must provide the operator *bool operator()(Point_2 p, Point_2 q)* which returns whether p is less than q according to their y-ordering.

Creation

Only a default constructor, copy constructor and an assignement operator are required. Note that further constructors can be provided.

LargestEmptyIsoRectangleTraits_2 traits; — Default constructor.

LargestEmptyIsoRectangleTraits_2 traits(LargestEmptyIsoRectangleTraits_2);

Copy constructor

LargestEmptyIsoRectangleTraits_2

traits = gtr — Assignment operator.

Predicate functions

The following functions give access to the predicate and constructor objects.

Compare_x_2	*traits.compare_x_2_object()*
Compare_y_2	*traits.compare_y_2_object()*
Less_x_2	*traits.less_x_2_object()*
Less_y_2	*traits.less_y_2_object()*

Has Models

CGAL::Cartesian<R>
CGAL::Homogeneous<R>

See Also

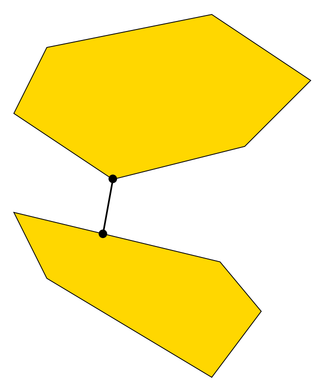
CGAL::Largest_empty_iso_rectangle_2<Traits>

Chapter 68

Optimal Distances

Kaspar Fischer, Bernd Gärtner, Thomas Herrmann, Michael Hoffmann, and Sven Schönherr

This chapter describes how to compute the distance between the convex hulls of two given point sets in d-dimensional Euclidean space (*CGAL::Polytope_distance_d<Traits>*). Moreover, it is possible to compute the width of a point set in three dimensions (*CGAL::Width_3<Traits>*).



The obvious application is collision detection between convex bodies in space. In the spirit of the bounding volume application above, it also makes sense for nonconvex objects: a full intersection test between complicated objects could in a first stage be approximated with the test between the convex hulls of the objects. Only if the hulls intersect, a full intersection test is necessary.

To dampen fears concerning the performance of the distance computation, we want to mention that the convex hulls of the input point sets are not explicitly computed. This avoids a runtime which grows exponentially in d. In fact, the runtime is almost always linear in the size of the two point sets.

Optimal Distances
Reference Manual

Kaspar Fischer, Bernd Gärtner, Thomas Herrmann, Michael Hoffmann, and Sven Schönherr

This chapter describes concepts, classes, and functions for 2D all furthest neigbors, 3D width of a point set, and dD polytope distance.

Assertions

The optimization code uses infix *OPTIMISATION* in the assertions, e.g. defining the compiler flag *CGAL_OPTIMISATION_NO_PRECONDITIONS* switches precondition checking off, cf. Section 2.8.

68.1 Classified References Pages

All furthest neighbors

Width

Polytope Distance

68.2 Alphabetical List of Reference Pages

CGAL::all_furthest_neighbors_2

Function

Definition

The function *all_furthest_neighbors_2* computes all furthest neighbors for the vertices of a convex polygon P, i.e. for each vertex v of P a vertex f_v of P such that the distance between v and f_v is maximized.

#include <CGAL/all_furthest_neighbors_2.h>

template < class RandomAccessIterator, class OutputIterator, class Traits >
OutputIterator all_furthest_neighbors_2(
RandomAccessIterator points_begin,
RandomAccessIterator points_end,
OutputIterator o,
Traits t = Default_traits)

computes all furthest neighbors for the vertices of the convex polygon described by the range [*points_begin*, *points_end*), writes their indices (relative to *points_begin*) to *o*[1] and returns the past-the-end iterator of this sequence.

Precondition: The points denoted by the non-empty range [*points_begin*, *points_end*) form the boundary of a convex polygon P (oriented clock– or counterclockwise).

The geometric types and operations to be used for the computation are specified by the traits class parameter t. This parameter can be omitted if *RandomAccessIterator* refers to a point type from a *Kernel*. In this case, the kernel is used as default traits class.

Requirement:

1. If t is specified explicitly, *Traits* is a model for *AllFurthestNeighborsTraits_2*.
2. Value type of *RandomAccessIterator* is *Traits::Point_2* or – if t is not specified explicitly – *K::Point_2* where K is a model for *Kernel*.
3. *OutputIterator* accepts *int* as value type.

See Also

AllFurthestNeighborsTraits_2 . . . . page 3935
CGAL::monotone_matrix_search . . . . page 458

Implementation

The implementation uses monotone matrix search[AKM+87]. Its runtime complexity is linear in the number of vertices of P.

[1] i.e. the furthest neighbor of *points_begin*[i] is *points_begin*[*i*-th number written to *o*]

Example

The following code generates a random convex polygon *p* with ten vertices, computes all furthest neighbors and writes the sequence of their indices (relative to *points_begin*) to *cout* (e.g. a sequence of *4788911224* means the furthest neighbor of *points_begin[0]* is *points_begin[4]*, the furthest neighbor of *points_begin[1]* is *points_begin[7]* etc.).

```
#include <CGAL/Cartesian.h>
#include <CGAL/Polygon_2.h>
#include <CGAL/point_generators_2.h>
#include <CGAL/random_convex_set_2.h>
#include <CGAL/all_furthest_neighbors_2.h>
#include <CGAL/IO/Ostream_iterator.h>
#include <iostream>
#include <vector>

typedef double                                          FT;

typedef CGAL::Cartesian<FT>                             Kernel;

typedef Kernel::Point_2                                 Point;
typedef std::vector<int>                                Index_cont;
typedef CGAL::Polygon_2<Kernel>                         Polygon_2;
typedef CGAL::Random_points_in_square_2<Point>          Generator;
typedef CGAL::Ostream_iterator<int,std::ostream>        Oiterator;

int main()
{
  // generate random convex polygon:
  Polygon_2 p;
  CGAL::random_convex_set_2(10, std::back_inserter(p), Generator(1));

  // compute all furthest neighbors:
  CGAL::all_furthest_neighbors_2(p.vertices_begin(), p.vertices_end(),
                                 Oiterator(std::cout));
  std::cout << std::endl;

  return 0;
}
```

File: examples/Matrix_search/all_furthest_neighbors_2.cpp

Concept

AllFurthestNeighborsTraits_2

Definition

The concept AllFurthestNeighborsTraits_2 defines types and operations needed to compute all furthest neighbors for the vertices of a convex polygon using the function *all_furthest_neighbors_2*.

Types

AllFurthestNeighborsTraits_2:: FT	model for FieldNumberType . . . . page 149.
AllFurthestNeighborsTraits_2:: Point_2	model for Kernel::Point_2 . . . . page 794.
AllFurthestNeighborsTraits_2:: Compute_squared_distance_2	
	model for Kernel::Compute_squared_distance_2 . . . page **??**.
AllFurthestNeighborsTraits_2:: Less_xy_2	model for Kernel::Less_xy_2 . . . . page **??**.
AllFurthestNeighborsTraits_2:: Orientation_2	model for Kernel::Orientation_2 . . . . page 1034.

Operations

The following member functions return function objects of the types listed above.

Compute_squared_distance_2	
	t.compute_squared_distance_2_object()
Less_xy_2	*t.less_xy_2_object()*
Orientation_2	*t.orientation_2_object()*

Has Models

Cartesian<FieldNumberType> . . . . page **??**, *Homogeneous<RingNumberType>* . . . . page **??**,
Simple_cartesian<FieldNumberType> . . . . . . page **??**, *Simple_homogeneous<RingNumberType>* . . . . . . page **??**.

See Also

CGAL::all_furthest_neighbors_2 . . . . page 3933

Notes

- *::Less_xy_2* and *::Orientation_2* are used for (expensive) precondition checking only. Therefore, they need not to be specified, in case that precondition checking is disabled.

CGAL::Width_3<Traits>

Definition

Given a set of points $\mathcal{S} = \{p_1, \ldots, p_n\}$ in $\mathbb{R}^3$. The width of $\mathcal{S}$, denoted as $\mathcal{W}(\mathcal{S})$, is defined as the minimum distance between two parallel planes of support of $conv(\mathcal{S})$; where $conv(\mathcal{S})$ denotes the convex hull of $\mathcal{S}$. The width in direction $\mathbf{d}$, denoted as $\mathcal{W}_d(\mathcal{S})$, is the distance between two parallel planes of support of $conv(\mathcal{S})$, which are orthogonal to $\mathbf{d}$.

Subject to the applications of the width algorithm, several objects might be interesting:

1. The two parallel planes of support such that the distance between them is as small as possible. These planes are called width-planes in further considerations.
2. The width $\mathcal{W}(\mathcal{S})$, i.e., the distance between the width-planes.
3. The direction $\mathbf{d}_{opt}$ such that $\mathcal{W}(\mathcal{S}) = \mathcal{W}_{d_{opt}}(\mathcal{S})$

Note: There might be several optimal build directions. Hence neither the width-planes nor the direction $\mathbf{d}_{opt}$ are unique – only the width is.

#include <CGAL/Width_3.h>

Requirements

The template parameter *Traits* is a model for *WidthTraits_3*. We provide the model *Width_default_traits_3< Kernel>* based on a three-dimensional CGAL kernel.

Types

Width_3<Traits>:: Traits		traits class.
typedef typename Traits::Point_3	*Point_3;*	point type.
typedef typename Traits::Plane_3	*Plane_3;*	plane type.
typedef typename Traits::Vector_3	*Vector_3;*	vector type.
typedef typename Traits::RT	*RT;*	algebraic ring type.
typedef typename Traits::ChullTraits	*ChullTraits;*	traits class for the 3D convex hull algorithm.

Creation

template < class InputIterator >
Width_3<Traits> width(InputIterator first, InputIterator beyond);

creates a variable *width* initialized to the width of $\mathcal{S}$ – with $\mathcal{S}$ being the set of points in the range [*first*,*beyond*).
Requirement: The value type of *InputIterator* is *Point_3*.

template < class Polyhedron >
Width_3<Traits> width(Polyhedron& P);

creates a variable *width* initialized to the width of the polyhedron P. Note that the vertex point coordinates are altered!
Precondition: P is a convex polyhedron.
Requirement: *Polyhedron* is a *CGAL::Polyhedron_3* with facets supporting plane equations where *Polyhedron::Point_3* $\equiv$ *Point_3* and *Polyhedron::Plane_3* $\equiv$ *Plane_3*.

Access Functions

void *width.get_squared_width(RT& width_num, RT& width_denom)*

returns the squared width. For the reason of exact computation not the width itself is stored, but the *squared* width as a fraction: The numerator in *width_num* and the denominator in *width_denom*. The width of the point set $\mathcal{S}$ is $\sqrt{\frac{width\_num}{width\_denom}}$.

void *width.get_width_planes(Plane_3& e1, Plane_3& e2)*

The planes *e1* and *e2* are the two parallel supporting planes, which distance is minimal (among all such planes).

void *width.get_width_coefficients(RT& A, RT& B, RT& C, RT& D, RT& K)*

The returned coefficients *A,B,C,D,K* have the property that width-plane *e1* is given by the equation $Ax+By+Cz+D=0$ and width-plane *e2* by $Ax+By+Cz+K=0$.

Vector_3 *width.get_build_direction()*

returns a direction $\mathbf{d}_{opt}$ such that the width-planes *e1* and *e2* are perpendicular to $\mathbf{d}_{opt}$. The width of the point set is minimal in this direction.

void *width.get_all_build_directions(std::vector<Vector_3>& dir)*

All the build directions are stored in the vector *dir*. It might happen that a certain body has several different build directions, but it is also possible to have only one build direction.

int *width.get_number_of_optimal_solutions()*

returns the number of optimal solutions, i.e., the number of optimal build directions.

See Also

CGAL::Width_default_traits_3<K> . . . . page 3940
WidthTraits_3 . . . . page 3942

Implementation

Since the width of the point set $\mathcal{S}$ and the width of the convex hull of $\mathcal{S}$ ($conv(\mathcal{S})$) is the same, the algorithm uses the 3D convex hull algorithm CGAL provides.

The width-algorithm is not incremental and therefore inserting and erasing points cause not an 'automatic' update of the width. Instead you have to run the width-algorithm again even if the point set is extended by only one new point.

advanced

Large Numbers. Because there is no need for dividing values during the algorithm, the numbers can get really huge (all the computations are made using a lot of multiplications). Therefore it is strongly recommended to use a number type that can handle numbers of arbitrary length (e.g., *leda_integer* in combination with the homogeneous representation of the points). But these large numbers have a disadvantage: Operations on them are slower as greater the number gets. Therefore it is possible to shorten the numbers by using the compiler flag -DSIMPLIFY. For using this option it is required that the underlying number type provides the 'modulo' operation.

Information Output during the Computations. If during the algorithm the program should output some information (e.g., during the debugging phase) you can turn on the output information by giving the compiler flag DEBUG. In the file `width_assertions.h` you can turn on/off the output of some functions and additional informations by changing the defined values from 0 (no output) to 1 (output available). But then it is required that the <<-operator has to been overloaded for *Point_3*, *Plane_3*, *Vector_3* and *RT*.

advanced

Example

```
#include <CGAL/Homogeneous.h>
#include <CGAL/Width_default_traits_3.h>
#include <CGAL/Width_3.h>
#include <iostream>
#include <vector>

#if defined(CGAL_USE_GMP)
#include <CGAL/Gmpz.h>
typedef CGAL::Gmpz                          RT;
#elif defined (CGAL_USE_LEDA)
#include <CGAL/leda_integer.h>
typedef leda_integer                         RT;
#else
#include <CGAL/MP_Float.h>
typedef CGAL::MP_Float                       RT;
#endif
```

```
typedef CGAL::Homogeneous<RT>                       Kernel;
typedef Kernel::Point_3                             Point_3;
typedef Kernel::Plane_3                             Plane_3;
typedef CGAL::Width_default_traits_3<Kernel>  Width_traits;
typedef CGAL::Width_3<Width_traits>                 Width;

int main() {
    // Create a simplex using homogeneous integer coordinates
    std::vector<Point_3> points;
    points.push_back( Point_3(2,0,0,1));
    points.push_back( Point_3(0,1,0,1));
    points.push_back( Point_3(0,0,1,1));
    points.push_back( Point_3(0,0,0,1));

    // Compute width of simplex
    Width simplex( points.begin(), points.end());

    // Output of squared width, width-planes, and optimal direction
    RT wnum, wdenom;
    simplex.get_squared_width( wnum, wdenom);
    std::cout << "Squared Width: " << wnum << "/" << wdenom << std::endl;

    std::cout << "Direction: " << simplex.get_build_direction() <<
   std::endl;

    Plane_3  e1, e2;
    std::cout << "Planes: E1: " << e1 << ".  E2: " << e2 <<std::endl;

    std::cout << "Number of optimal solutions: "
              << simplex.get_number_of_optimal_solutions() << std::endl;
    return(0);
}
```

File: examples/Width_3/width_simplex.cpp

Class

CGAL::Width_default_traits_3<K>

Definition

The class *Width_default_traits_3<K>* is a traits class for *Width_3<Traits>* using the three-dimensional CGAL kernel.

#include <CGAL/Width_default_traits_3.h>

Requirements

The template parameter *K* is a model for *Kernel*

Is Model for the Concepts

WidthTraits_3 . . . . page 3942

Types

typedef typename K::Point_3	*Point_3;*
typedef typename K::Plane_3	*Plane_3;*
typedef typename K::Vector_3	*Vector_3;*
typedef typename K::RT	*RT;*
typedef Convex_hull_traits_3<K>	*ChullTraits;*

Creation

Width_default_traits_3<K> traits; default constructor.

Operations

RT *traits.get_hx(Point_3 p) const*
returns the homogeneous x-coordinate of point p.

RT *traits.get_hy(Point_3 p) const*
returns the homogeneous y-coordinate of point p.

RT *traits.get_hz(Point_3 p) const*
returns the homogeneous z-coordinate of point p.

RT *traits.get_hw(Point_3 p) const*
returns the homogenizing coordinate of point p.

void *traits.get_point_coordinates(Point_3 p, RT& px, RT& py, RT& pz, RT& ph) const*
returns all homogeneous coordinates of point p at once.

RT *traits.get_a(Plane_3 f) const*
returns the first coefficient of plane f.

RT *traits.get_b(Plane_3 f) const*
returns the second coefficient of plane f.

RT	*traits.get_c(Plane_3 f) const*	
		returns the third coefficient of plane f.
RT	*traits.get_d(Plane_3 f) const*	
		returns the fourth coefficient of plane f.
void	*traits.get_plane_coefficients(Plane_3 f, RT& a, RT& b, RT& c, RT& d) const*	
		returns all four plane coefficients of f at once.
Point_3	*traits.make_point(RT hx, RT hy, RT hz, RT hw) const*	
		returns a point of type *Point_3* with homogeneous coordinates *hx*, *hy*, *hz* and *hw*.
Plane_3	*traits.make_plane(RT a, RT b, RT c, RT d) const*	
		returns a plane of type *Plane_3* whose coefficients are *a*, *b*, *c* and *d*.
Vector	*traits.make_vector(RT a, RT b, RT c) const*	
		returns a vector of type *Vector_3* with the four coefficients *a*, *b*, *c* and 1.

See Also

CGAL::Width_3<Traits> . . . . page 3936
WidthTraits_3 . . . . page 3942

Concept

WidthTraits_3

Definition

This concept defines the requirements for traits classes of *Width_3<Traits>*.

Types

WidthTraits_3:: Point_3	The point type. The (in)equality tests must be available. Access to the point coordinates is done via the *get_()* functions. Constructing a point is done with the *make_point( )* operation.
WidthTraits_3:: Plane_3	The plane type. Access to the coefficients of the plane is made via the *get_()* functions. Constructing a plane is done with the *make_plane()* operation.
WidthTraits_3:: Vector_3	The vector type. There is no need to access the coefficients of a vector; only constructing is required and is done with the *make_vector* operation.
WidthTraits_3:: ChullTraits	The traits class for using the convex hull algorithm. It must be a model of the concept ConvexHullTraits_3. This class is used only if the width is computed from a set of points.
WidthTraits_3:: RT	Ring type numbers. Internally all numbers are treated as ring type numbers, i.e., neither $/$-operator nor $\sqrt{.}$ nor other inexact operations are used. But because the algorithm does not use any divisions, but multiplication instead, the numbers can get really big. Therefore it is recommended to use a ring type number, that provides values of arbitrary length. Furthermore it is assumed that the underlying number type of *Point_3*, *Plane_3* and *Vector_3* equals *RT*.

Notes: If you want to compute the width of a *polyhedron* then you have to make sure that the point type in the traits class and the point type in the polyhedron class are the same! The same holds for *Traits::Plane_3* and *Polyhedron::Plane_3*.

Creation

Only a default constructor is required.

WidthTraits_3 traits;

Operations

Whatever the coordinates of the points are, it is required for the width-algorithm to have access to the homogeneous representation of points.

RT	*get_hx(Point_3 p) const*	returns the homogeneous x-coordinate of point p.
RT	*get_hy(Point_3 p) const*	returns the homogeneous y-coordinate of point p.
RT	*get_hz(Point_3 p) const*	returns the homogeneous z-coordinate of point p.
RT	*get_hw(Point_3 p) const*	returns the homogenizing coordinate of point p.
void	*get_point_coordinates(Point_3 p, RT& px, RT& py, RT& pz, RT& ph) const*	returns all homogeneous coordinates of point p at once.
RT	*get_a(Plane_3 f) const*	returns the first coefficient of plane f.
RT	*get_b(Plane_3 f) const*	returns the second coefficient of plane f.

RT	*get_c(Plane_3 f) const*	returns the third coefficient of plane f.
RT	*get_d(Plane_3 f) const*	returns the fourth coefficient of plane f.
void	*get_plane_coefficients(Plane_3 f, RT& a, RT& b, RT& c, RT& d) const*	returns all four plane coefficients of f at once.
Point_3	*make_point(RT hx, RT hy, RT hz, RT hw) const*	returns a point of type *Point_3* with homogeneous coordinates hx, hy, hz and hw.
Plane_3	*make_plane(RT a, RT b, RT c, RT d) const*	returns a plane of type *Plane_3* with coefficients a, b, c and d.
Vector_3	*make_vector(RT a, RT b, RT c) const*	returns a vector of type *Vector_3* with the four homogeneous coefficients a, b, c and 1.

Has Models

CGAL::Width_default_traits_3<K> . . . . page 3940

See Also

CGAL::Width_3<Traits> . . . . page 3936

Class

CGAL::Polytope_distance_d<Traits>

Definition

An object of the class *Polytope_distance_d<Traits>* represents the (squared) distance between two convex polytopes, given as the convex hulls of two finite point sets in d-dimensional Euclidean space $\mathbb{E}^d$. For point sets P and Q we denote by $pd(P,Q)$ the distance between the convex hulls of P and Q. Note that $pd(P,Q)$ can be degenerate, i.e. $pd(P,Q) = \infty$ if P or Q is empty.

Two inclusion-minimal subsets S_P of P and S_Q of Q with $pd(S_P,S_Q) = pd(P,Q)$ are called *pair of support sets*, the points in S_P and S_Q are the *support points*. A pair of support sets has size at most $d+2$ (by size we mean $|S_P| + |S_Q|$). The distance between the two polytopes is *realized* by a pair of points p and q lying in the convex hull of S_P and S_Q, repectively, i.e. $\sqrt{||p-q||} = pd(P,Q)$. In general, neither the support sets nor the realizing points are necessarily unique.

The underlying algorithm can cope with all kinds of input, e.g. P and Q may be in non-convex position or points may occur more than once. The algorithm computes a pair of support sets S_P and S_Q with realizing points p and q which remain fixed until the next set, insert, or clear operation.

#include <CGAL/Polytope_distance_d.h>

Requirements

The template parameter *Traits* is a model for *OptimisationDTraits*.

We provide the models *Optimisation_d_traits_2*, *Optimisation_d_traits_3*, and *Optimisation_d_traits_d* using the two-, three-, and d-dimensional CGAL kernel, respectively.

Types

Polytope_distance_d<Traits>:: Point	typedef to *Traits::Point_d*. Point type used to represent the input points.
Polytope_distance_d<Traits>:: FT	typedef to *Traits::FT*. Number type used to return the squared distance between the two polytopes.
Polytope_distance_d<Traits>:: ET	typedef to *Traits::ET*. Number type used to do the exact computations in the underlying solver for quadratic programs (cf. **Implementation**).

Polytope_distance_d<Traits>:: Point_iterator

non-mutable model of the STL concept *RandomAccessIterator* with value type *Point*. Used to access the points of the two polytopes.

Polytope_distance_d<Traits>:: Support_point_iterator

non-mutable model of the STL concept *RandomAccessIterator* with value type *Point*. Used to access the support points.

Polytope_distance_d<Traits>:: Support_point_index_iterator

non-mutable model of the STL concept *RandomAccessIterator* with value type *int*. Used to access the indices of the support points in the provided input order (starting from 0 in both point sets).

Polytope_distance_d<Traits>:: Coordinate_iterator

non-mutable model of the STL concept *RandomAccessIterator* with value type *ET*. Used to access the coordinates of the realizing points.

Creation

Polytope_distance_d<Traits> poly_dist(Traits traits = Traits(), int verbose = 0, std::ostream& stream = std::cout)

initializes *poly_dist* to $pd(\emptyset,\emptyset)$.

template < class InputIterator1, class InputIterator2 >
Polytope_distance_d<Traits> poly_dist(InputIterator1 p_first, InputIterator1 p_last, InputIterator2 q_first, InputIterator2 q_last, Traits traits = Traits(), int verbose = 0, std::ostream& stream = std::cout)

initializes *poly_dist* to $pd(P,Q)$ with P and Q being the sets of points in the range [*p_first*,*p_last*) and [*q_first*,*q_last*), respectively.
Requirement: The value type of *InputIterator1* and *InputIterator2* is *Point*.
Precondition: All points have the same dimension.

advanced

If *verbose* is set to 1, 2, or 3 then some, more, or full verbose output of the underlying solver for quadratic programs is written to *stream*, resp.

advanced

Access Functions

int *poly_dist.ambient_dimension() const*

returns the dimension of the points in P and Q. If *poly_dist* is $pd(\emptyset,\emptyset)$, the ambient dimension is -1.

int *poly_dist.number_of_points() const*
returns the number of all points of *poly_dist*, i.e. $|P| + |Q|$.

int *poly_dist.number_of_points_p() const*
returns the number of points in P.

int *poly_dist.number_of_points_q() const*
returns the number of points in Q.

int *poly_dist.number_of_support_points() const*
returns the number of support points of *poly_dist*, i.e. $|S_P| + |S_Q|$.

int *poly_dist.number_of_support_points_p() const*
returns the number of support points in S_P.

int *poly_dist.number_of_support_points_q() const*
returns the number of support points in S_Q.

Point_iterator *poly_dist.points_p_begin() const*
returns an iterator referring to the first point in P.

Point_iterator *poly_dist.points_p_end() const*
returns the corresponding past-the-end iterator.

Point_iterator *poly_dist.points_q_begin() const*
returns an iterator referring to the first point in Q.

Point_iterator *poly_dist.points_q_end() const*
returns the corresponding past-the-end iterator.

Support_point_iterator
poly_dist.support_points_p_begin() const
returns an iterator referring to the first support point in S_P.

Support_point_iterator
poly_dist.support_points_p_end() const
returns the corresponding past-the-end iterator.

Support_point_iterator
poly_dist.support_points_q_begin() const
returns an iterator referring to the first support point in S_Q.

Support_point_iterator
poly_dist.support_points_q_end() const
returns the corresponding past-the-end iterator.

Support_point_index_iterator
poly_dist.support_points_p_indices_begin() const
returns an iterator referring to the index of the first support point in P.

Support_point_index_iterator
poly_dist.support_points_p_indices_end() const
returns the corresponding past-the-end iterator.

Support_point_index_iterator
poly_dist.support_points_q_indices_begin() const
returns an iterator referring to the index of the first support point in Q.

Support_point_index_iterator

poly_dist.support_points_q_indices_end() const

returns the corresponding past-the-end iterator.

Point *poly_dist.realizing_point_p() const*

returns the realizing point of P.
Requirement: An implicit conversion from *ET* to *RT* is available.
Precondition: $pd(P,Q)$ is finite.

Point *poly_dist.realizing_point_q() const*

returns the realizing point of Q.
Requirement: An implicit conversion from *ET* to *RT* is available.
Precondition: $pd(P,Q)$ is finite.

FT *poly_dist.squared_distance() const*

returns the squared distance of *poly_dist*, i.e. $(pd(P,Q))^2$.
Requirement: An implicit conversion from *ET* to *RT* is available.
Precondition: $pd(P,Q)$ is finite.

Coordinate_iterator *poly_dist.realizing_point_p_coordinates_begin() const*

returns an iterator referring to the first coordinate of the realizing point of P.
Note: The coordinates have a rational representation, i.e. the first d elements of the iterator range are the numerators and the $(d+1)$-st element is the common denominator.

Coordinate_iterator *poly_dist.realizing_point_p_coordinates_end() const*

returns the corresponding past-the-end iterator.

Coordinate_iterator *poly_dist.realizing_point_q_coordinates_begin() const*

returns an iterator referring to the first coordinate of the realizing point of Q.
Note: The coordinates have a rational representation, i.e. the first d elements of the iterator range are the numerators and the $(d+1)$-st element is the common denominator.

Coordinate_iterator *poly_dist.realizing_point_q_coordinates_end() const*

returns the corresponding past-the-end iterator.

ET *poly_dist.squared_distance_numerator() const*

returns the numerator of the squared distance of *poly_dist*.

ET *poly_dist.squared_distance_denominator() const*

returns the denominator of the squared distance of *poly_dist*.

Predicates

bool *poly_dist.is_finite() const*

returns *true*, if $pd(P,Q)$ is finite, i.e. none of the two polytopes is empty.

bool *poly_dist.is_zero() const*

returns *true*, if $pd(P,Q)$ is zero, i.e. the two polytopes intersect (this implies degeneracy).

bool *poly_dist.is_degenerate() const*

returns *true*, iff $pd(P,Q)$ is degenerate, i.e. $pd(P,Q)$ is not finite.

Modifiers

void *poly_dist.clear()* resets *poly_dist* to $pd(\emptyset,\emptyset)$.

template < class InputIterator1, class InputIterator2 >
void *poly_dist.set(InputIterator1 p_first,*
InputIterator1 p_last,
InputIterator2 q_first,
InputIterator2 q_last)

sets *poly_dist* to $pd(P,Q)$ with P and Q being the sets of points in the ranges [*p_first*,*p_last*) and [*q_first*,*q_last*), respectively.
Requirement: The value type of *InputIterator1* and *InputIterator2* is *Point*.
Precondition: All points have the same dimension.

template < class InputIterator >
void *poly_dist.set_p(InputIterator p_first, InputIterator p_last)*

sets *poly_dist* to $pd(P,Q)$ with P being the set of points in the range [*p_first*,*p_last*) (Q remains unchanged).
Requirement: The value type of *InputIterator* is *Point*.
Precondition: All points in P have dimension *poly_dist.ambient_dimension()* if Q is not empty.

template < class InputIterator >
void *poly_dist.set_q(InputIterator q_first, InputIterator q_last)*

sets *poly_dist* to $pd(P,Q)$ with Q being the set of points in the range [*q_first*,*q_last*) (P remains unchanged).
Requirement: The value type of *InputIterator* is *Point*.
Precondition: All points in Q have dimension *poly_dist.ambient_dimension()* if P is not empty.

void *poly_dist.insert_p(Point p)*

inserts p into P.
Precondition: The dimension of p equals *poly_dist.ambient_dimension()* if *poly_dist* is not $pd(\emptyset,\emptyset)$.

void *poly_dist.insert_q(Point q)*

inserts q into Q.
Precondition: The dimension of q equals *poly_dist.ambient_dimension()* if *poly_dist* is not $pd(\emptyset,\emptyset)$.

template < class InputIterator1, class InputIterator2 >
void *poly_dist.insert(InputIterator1 p_first, InputIterator1 p_last, InputIterator2 q_first, InputIterator2 q_last)*

inserts the points in the range [*p_first*,*p_last*) and [*q_first*,*q_last*) into P and Q, respectively, and recomputes the (squared) distance.
Requirement: The value type of *InputIterator1* and *InputIterator2* is *Point*.
Precondition: All points have the same dimension. If *poly_dist* is not $pd(\emptyset,\emptyset)$, this dimension must be equal to *poly_dist.ambient_dimension()*.

template < class InputIterator >
void *poly_dist.insert_p(InputIterator p_first, InputIterator p_last)*

inserts the points in the range [*p_first*,*p_last*) into P and recomputes the (squared) distance (Q remains unchanged).
Requirement: The value type of *InputIterator* is *Point*.
Precondition: All points have the same dimension. If *poly_dist* is not empty, this dimension must be equal to *poly_dist.ambient_dimension()*.

template < class InputIterator >
void *poly_dist.insert_q(InputIterator q_first, InputIterator q_last)*

inserts the points in the range [*q_first*,*q_last*) into Q and recomputes the (squared) distance (P remains unchanged).
Requirement: The value type of *InputIterator* is *Point*.
Precondition: All points have the same dimension. If *poly_dist* is not empty, this dimension must be equal to *poly_dist.ambient_dimension()*.

Validity Check

An object *poly_dist* is valid, iff ...

- *poly_dist* contains all points of its defining set *P*,
- *poly_dist* is the smallest sphere containing its support set *S*, and
- *S* is minimal, i.e. no support point is redundant.

bool *poly_dist.is_valid(bool verbose = false, int level = 0) const*

returns *true*, iff *poly_dist* is valid. If *verbose* is *true*, some messages concerning the performed checks are written to standard error stream. The second parameter *level* is not used, we provide it only for consistency with interfaces of other classes.

Miscellaneous

const Traits& *poly_dist.traits() const*

returns a const reference to the traits class object.

I/O

std::ostream& *std::ostream& os << poly_dist*

writes *poly_dist* to output stream *os*.
Requirement: The output operator is defined for *Point_d*.

std::istream& *std::istream& is >> poly_dist&*

reads *poly_dist* from input stream *is*.
Requirement: The input operator is defined for *Point_d*.

See Also

CGAL::Optimisation_d_traits_2<K,ET,NT> . . . . page **??**
CGAL::Optimisation_d_traits_3<K,ET,NT> . . . . page **??**
CGAL::Optimisation_d_traits_d<K,ET,NT> . . . . page **??**

OptimisationDTraits . . . . page **??**

Implementation

The problem of finding the distance between two convex polytopes given as the convex hulls of two finite point sets can be formulated as an optimization problem with linear constraints and a convex quadratic objective function. The solution is obtained using our exact solver for quadratic programs [GS00].

The creation time is almost always linear in the number of points. Access functions and predicates take constant time, inserting a point might take up to linear time. The clear operation and the check for validity each take linear time.

PolytopeDistanceDTraits

Definition

This concept defines the requirements for traits classes of d-dimensional optimisation algorithms.

Types

PolytopeDistanceDTraits:: Point_d — point type used to represent the input points.

PolytopeDistanceDTraits:: Rep_tag — compile time tag to distinguish between Cartesian and homogeneous representation of the input points. *Rep_tag* has to be either *CGAL::Cartesian_tag* or *CGAL::Homogeneous_tag*.

PolytopeDistanceDTraits:: RT — number type used to represent the coordinates of the input points. It has to be a model for *RingNumberType*.

PolytopeDistanceDTraits:: FT — number type used to return either the squared radius of the smallest enclosing sphere or annulus, or the squared distance of the polytopes. *FT* has to be either *RT* or *CGAL::Quotient*<*RT*> if the input points have Cartesian or homogeneous representation, respectively (cf. *Rep_tag*).

PolytopeDistanceDTraits:: Access_dimension_d

data accessor object used to access the dimension of the input points.

PolytopeDistanceDTraits:: Access_coordinates_begin_d

data accessor object used to access the coordinates of the input points.

PolytopeDistanceDTraits:: Construct_point_d

constructor object used to construct either the center of the smallest enclosing sphere or annulus, or the points realizing the distance between the two polytopes.

The following two number types are only needed for *CGAL::Min_annulus_d*<*Traits*> and *CGAL::Polytope_distance_d*<*Traits*>.

PolytopeDistanceDTraits:: ET — exact number type used to do the exact computations in the underlying solver for linear programs. It has to to be a model for *RingNumberType*. There must be an implicit conversion from *RT* to *ET* available.

PolytopeDistanceDTraits:: NT — fast (possibly inexact) number type used to speed up the pricing step in the underlying solver for linear programs. It has to be a model for *RingNumberType*. There must be implicit conversions from *RT* to *NT* and from *NT* to *ET* available.

Creation

Only default and copy constructor are required.

PolytopeDistanceDTraits traits;

PolytopeDistanceDTraits traits(PolytopeDistanceDTraits);

Operations

The following functions just return the corresponding function class object.

Access_dimension_d	*traits.access_dimension_d_object() const*
Access_coordinates_begin_d	*traits.access_coordinates_begin_d_object() const*
Construct_point_d	*traits.construct_point_d_object() const*

Has Models

CGAL::Polytope_distance_d_traits_2<K,ET,NT> . . . . page 3953
CGAL::Polytope_distance_d_traits_3<K,ET,NT> . . . . page 3955
CGAL::Polytope_distance_d_traits_d<K,ET,NT> . . . . page 3957

See Also

CGAL::Polytope_distance_d<Traits> . . . . page 3944

CGAL::Polytope_distance_d_traits_2<K,ET,NT>

Class

Definition

The class *Polytope_distance_d_traits_2<K,ET,NT>* is a traits class for the d-dimensional optimisation algorithms using the two-dimensional CGAL kernel.

#include <CGAL/Polytope_distance_d_traits_2.h>

Requirements

The template parameter *K* is a model for *Kernel*. Template parameters *ET* and *NT* are models for *RingNumberType*.

The second and third template parameter have default type *K::RT*.

Is Model for the Concepts

PolytopeDistanceDTraits . . . . page 3951

Types

Polytope_distance_d_traits_2<K,ET,NT>:: Point_d typedef to *K::Point_2*.

Polytope_distance_d_traits_2<K,ET,NT>:: Rep_tag typedef to *K::Rep_tag*.

Polytope_distance_d_traits_2<K,ET,NT>:: RT typedef to *K::RT*.

Polytope_distance_d_traits_2<K,ET,NT>:: FT typedef to *K::FT*.

Polytope_distance_d_traits_2<K,ET,NT>:: Access_dimension_d

typedef to *K::Access_dimension_2*.

Polytope_distance_d_traits_2<K,ET,NT>:: Access_coordinates_begin_d

typedef to *K::Access_coordinates_begin_2*.

Polytope_distance_d_traits_2<K,ET,NT>:: Construct_point_d

typedef to *K::Construct_point_2*.

Polytope_distance_d_traits_2<K,ET,NT>:: ET second template parameter (default is *K::RT*).

Polytope_distance_d_traits_2<K,ET,NT>:: NT third template parameter (default is *K::RT*).

Creation

Polytope_distance_d_traits_2<K,ET,NT> traits; default constructor.

Polytope_distance_d_traits_2<K,ET,NT> traits(Polytope_distance_d_traits_2<K,ET,NT>);

copy constructor.

Operations

The following functions just return the corresponding function class object.

Access_dimension_d *traits.access_dimension_d_object() const*

Access_coordinates_begin_d *traits.access_coordinates_begin_d_object() const*

Construct_point_d *traits.construct_point_d_object() const*

See Also

CGAL::Polytope_distance_d<Traits> . . . . page 3944
CGAL::Polytope_distance_d_traits_3<K,ET,NT> . . . . page 3955
CGAL::Polytope_distance_d_traits_d<K,ET,NT> . . . . page 3957
PolytopeDistanceDTraits . . . . page 3951

CGAL::Polytope_distance_d_traits_3<K,ET,NT>

Class

Definition

The class *Polytope_distance_d_traits_3<K,ET,NT>* is a traits class for the d-dimensional optimisation algorithms using the three-dimensional CGAL kernel.

#include <CGAL/Polytope_distance_d_traits_3.h>

Requirements

The template parameter *K* is a model for *Kernel*. Template parameters *ET* and *NT* are models for *RingNumberType*.

The second and third template parameter have default type *K::RT*.

Is Model for the Concepts

PolytopeDistanceDTraits . . . . page 3951

Types

Polytope_distance_d_traits_3<K,ET,NT>:: Point_d typedef to *K::Point_3*.

Polytope_distance_d_traits_3<K,ET,NT>:: Rep_tag typedef to *K::Rep_tag*.

Polytope_distance_d_traits_3<K,ET,NT>:: RT typedef to *K::RT*.

Polytope_distance_d_traits_3<K,ET,NT>:: FT typedef to *K::FT*.

Polytope_distance_d_traits_3<K,ET,NT>:: Access_dimension_d

typedef to *K::Access_dimension_3*.

Polytope_distance_d_traits_3<K,ET,NT>:: Access_coordinates_begin_d

typedef to *K::Access_coordinates_begin_3*.

Polytope_distance_d_traits_3<K,ET,NT>:: Construct_point_d

typedef to *K::Construct_point_3*.

Polytope_distance_d_traits_3<K,ET,NT>:: ET second template parameter (default is *K::RT*).

Polytope_distance_d_traits_3<K,ET,NT>:: NT third template parameter (default is *K::RT*).

Creation

Polytope_distance_d_traits_3<K,ET,NT> traits; default constructor.

Polytope_distance_d_traits_3<K,ET,NT> traits(Polytope_distance_d_traits_3<K,ET,NT>);

copy constructor.

Operations

The following functions just return the corresponding function class object.

Access_dimension_d *traits.access_dimension_d_object() const*

Access_coordinates_begin_d *traits.access_coordinates_begin_d_object() const*

Construct_point_d *traits.construct_point_d_object() const*

See Also

CGAL::Polytope_distance_d<Traits> . . . . page 3944

CGAL::Polytope_distance_d_traits_2<K,ET,NT> . . . . page 3953
CGAL::Polytope_distance_d_traits_d<K,ET,NT> . . . . page 3957

PolytopeDistanceDTraits . . . . page 3951

CGAL::Polytope_distance_d_traits_d<K,ET,NT>

Class

Definition

The class *Polytope_distance_d_traits_d<K,ET,NT>* is a traits class for the d-dimensional optimisation algorithms using the d-dimensional CGAL kernel.

#include <CGAL/Polytope_distance_d_traits_d.h>

Requirements

The template parameter *K* is a model for *Kernel*. Template parameters *ET* and *NT* are models for *RingNumberType*.

The second and third template parameter have default type *K::RT*.

Is Model for the Concepts

PolytopeDistanceDTraits . . . . page 3951

Types

Polytope_distance_d_traits_d<K,ET,NT>:: Point_d — typedef to *K::Point_d*.

Polytope_distance_d_traits_d<K,ET,NT>:: Rep_tag — typedef to *K::Rep_tag*.

Polytope_distance_d_traits_d<K,ET,NT>:: RT — typedef to *K::RT*.

Polytope_distance_d_traits_d<K,ET,NT>:: FT — typedef to *K::FT*.

Polytope_distance_d_traits_d<K,ET,NT>:: Access_dimension_d

typedef to *K::Access_dimension_d*.

Polytope_distance_d_traits_d<K,ET,NT>:: Access_coordinates_begin_d

typedef to *K::Access_coordinates_begin_d*.

Polytope_distance_d_traits_d<K,ET,NT>:: Construct_point_d

typedef to *K::Construct_point_d*.

Polytope_distance_d_traits_d<K,ET,NT>:: ET — second template parameter (default is *K::RT*).

Polytope_distance_d_traits_d<K,ET,NT>:: NT — third template parameter (default is *K::RT*).

Creation

Polytope_distance_d_traits_d<K,ET,NT> traits; default constructor.

Polytope_distance_d_traits_d<K,ET,NT> traits(Polytope_distance_d_traits_d<K,ET,NT>);

copy constructor.

Operations

The following functions just return the corresponding function class object.

Access_dimension_d *traits.access_dimension_d_object() const*

Access_coordinates_begin_d *traits.access_coordinates_begin_d_object() const*

Construct_point_d *traits.construct_point_d_object() const*

See Also

CGAL::Polytope_distance_d<Traits> . . . . page 3944

CGAL::Polytope_distance_d_traits_2<K,ET,NT> . . . . page 3953
CGAL::Polytope_distance_d_traits_3<K,ET,NT> . . . . page 3955

PolytopeDistanceDTraits . . . . page 3951

Chapter 69

Principal Component Analysis

Pierre Alliez, Sylvain Pion and Ankit Gupta

Contents

This package provides functions to analyze sets of objects in 2D and 3D. It provides the computation of axis-aligned bounding boxes, centers of mass and principal component analysis for all bounded objects, as well as barycenters for sets of weighted points. Note that unlike most of the CGAL packages, this package uses approximation methods (in particular for linear least squares fitting) and is not intended to provide an exact canonical result in any sense.

69.1 Definitions

A *bounding box* for a set of objects is a cuboid that contains the set. An *axis-aligned bounding box* captures the maximum extents of all objects from the set within their coordinate system, i.e., a bounding box aligned with the axes of the coordinate system. Axis-aligned bounding boxes are frequently used in geometric algorithms as an indication of the general location of a data set, for either display, first-approximation spatial query, or spatial indexing purposes.

A *centroid* of a set of objects is their center of mass, i.e., the point whose coordinates are computed by means of coordinates of all points composing the objects. Note that although the general definition of center of mass incorporates a density function (and hence weighted means), the current implementation assumes a uniform density (see barycenter below defined for weighted points). For a point set $\{X_1, X_2, \ldots, X_N\}$ the centroid $\bar{X}$ is computed as

$$\bar{X} = \frac{1}{N}\sum_{i=1}^{N} X_i.$$

For a set of segments $\{S_1, S_2, ..., S_N\}$ the centroid $\bar{X}$ is computed as

$$\bar{X} = \frac{1}{\sum_{i=1}^{N}|S_i|}\sum_{i=1}^{N}|S_i|\,\bar{S}_i,$$

where $|S_i|$ stands for the length of a segment and $\bar{S}_i$ stands for its uniform barycenter (midpoint). For a set of triangles $\{T_1, T_2, ..., T_N\}$ the centroid $\bar{X}$ is computed as

$$\bar{X} = \frac{1}{\sum_{i=1}^{N}|T_i|}\sum_{i=1}^{N}|T_i|\,\bar{T}_i,$$

where $|T_i|$ stands for the area of a triangle and $\bar{T}_i$ stands for its uniform barycenter. Such definition still holds for more general objects where the Lebesgue measure (length in 1D, area in 2D, volume in 3D) is used for weighting the object barycenters. Centers of mass are used to summarize data sets for either approximation, spatial query or spatial indexing purposes.

A *barycenter* of a set of weighted points is the point whose coordinates are computed by means of weighted coordinates of all weighted points from the set. When all weights are equal the barycenter coincides with the centroid.

Given a set of objects, *linear least squares fitting* amounts to finding the linear sub-space which minimizes the sum of squared distances from all points composing the objects of the set, to their projection onto this linear sub-space. Such linear sub-space is obtained by so-called principal component analysis (PCA). PCA is defined as a transformation that transforms the objects to a new coordinate system such that the greatest variance by orthogonal projection of the objects comes to lie on the first coordinate (called the first principal component), the second greatest variance on the second coordinate, and so on. PCA is often used in geometric applications to reduce data sets to lower dimensions for analysis or approximation. Figure below illustrates (least squares) fitting of a line to a 2D point set, fitting of a line and a plane to a 3D point set and fitting of a plane to a set of 3D triangles.

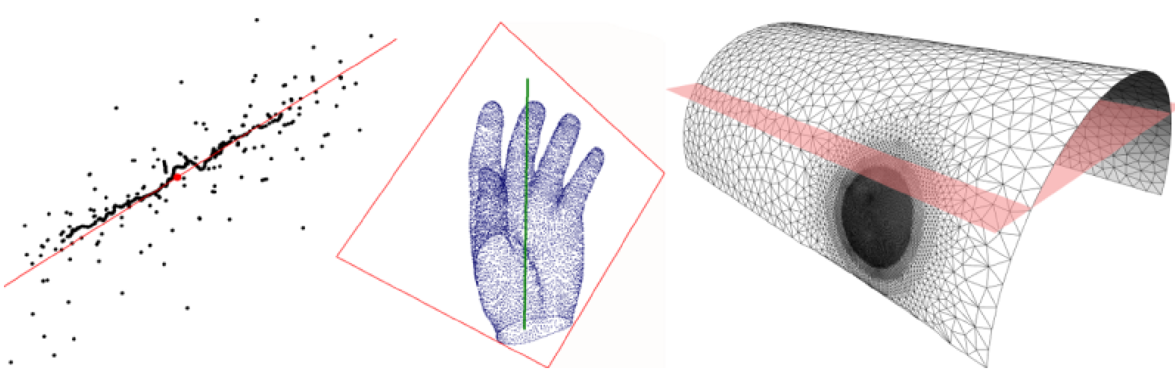


Figure 69.1: Left: fitting a line to a 2D point set (centroid dotted in red). Middle: fitting a line and a plane to a 3D point set. Right: fitting a plane to a set of 3D triangles.

69.2 Examples

69.2.1 Bounding Box of 2D and 3D Point Sets

In the following example we use STL containers of 2D points and 3D points, and compute their axis-aligned bounding box.

```
#include <CGAL/Cartesian.h>
#include <CGAL/bounding_box.h>

#include <list>
#include <iostream>

typedef double             FT;
typedef CGAL::Cartesian<FT> K;
typedef K::Point_2          Point_2;
typedef K::Point_3          Point_3;

int main()
{
  // axis-aligned bounding box of 2D points
  std::list<Point_2> points_2;
  points_2.push_back(Point_2(1.0, 0.0));
  points_2.push_back(Point_2(2.0, 2.0));
  points_2.push_back(Point_2(3.0, 5.0));

  K::Iso_rectangle_2 c2 = CGAL::bounding_box(points_2.begin(),
   points_2.end());
  std::cout << c2 << std::endl;

  // axis-aligned bounding box of 3D points
  std::list<Point_3> points_3;
  points_3.push_back(Point_3(1.0, 0.0, 0.5));
  points_3.push_back(Point_3(2.0, 2.0, 1.2));
  points_3.push_back(Point_3(3.0, 5.0, 4.5));

  K::Iso_cuboid_3 c3 = CGAL::bounding_box(points_3.begin(),
   points_3.end());
  std::cout << c3 << std::endl;

  return 0;
}
```

File: examples/Principal_component_analysis/bounding_box.cpp

69.2.2 Centroid of 2D and 3D Objects

In the following example we use STL containers of 2D, 3D points and 3D triangles, and compute their centroid. The dimension tag indicates the dimension of the objects being fitted (0 for points, 2 for triangles). Note that it is also possible to compute the centroid of the triangle edges (3D segments) by specifying 1 as dimension tag, and to compute the centroid of the triangle vertices (3D points) by specifying 0 as dimension tag.

```
#include <CGAL/Cartesian.h>
#include <CGAL/centroid.h>

#include <list>
#include <iostream>

typedef double                    FT;
typedef CGAL::Cartesian<FT>  K;
typedef K::Point_2           Point_2;
typedef K::Point_3           Point_3;
typedef K::Triangle_3        Triangle_3;

int main()
{
  // centroid of 2D points
  std::list<Point_2> points_2;
  points_2.push_back(Point_2(1.0, 0.0));
  points_2.push_back(Point_2(2.0, 2.0));
  points_2.push_back(Point_2(3.0, 5.0));
  Point_2 c2 = CGAL::centroid(points_2.begin(),
   points_2.end(),CGAL::Dimension_tag<0>());
  std::cout << c2 << std::endl;

  // centroid of 3D points
  std::list<Point_3> points_3;
  points_3.push_back(Point_3(1.0, 0.0, 0.5));
  points_3.push_back(Point_3(2.0, 2.0, 1.2));
  points_3.push_back(Point_3(3.0, 5.0, 4.5));
  Point_3 c3 = CGAL::centroid(points_3.begin(),
   points_3.end(),CGAL::Dimension_tag<0>());
  std::cout << c3 << std::endl;

  // centroid of 3D triangles
  std::list<Triangle_3> triangles_3;
  Point_3 p(1.0, 0.0, 0.0);
  Point_3 q(1.0, 2.0, 0.0);
  Point_3 r(0.0, 1.0, 3.0);
  Point_3 s(0.0, 2.0, 5.0);
  triangles_3.push_back(Triangle_3(p,q,r));
  triangles_3.push_back(Triangle_3(p,q,s));
  c3 = CGAL::centroid(triangles_3.begin(),
   triangles_3.end(),CGAL::Dimension_tag<2>());
  std::cout << c3 << std::endl;

  return 0;
}
```

File: examples/Principal_component_analysis/centroid.cpp

69.2.3 Barycenter of a Set of Weighted Points

In the following example we use STL containers of 2D and 3D weighted points, and compute their barycenter.

```
#include <CGAL/Cartesian.h>
#include <CGAL/barycenter.h>

#include <list>
#include <iostream>
#include <utility>

typedef double                   FT;
typedef CGAL::Cartesian<FT>  K;
typedef K::Point_2               Point_2;
typedef K::Point_3               Point_3;

int main()
{
  // barycenter of 2D weighted points
  std::list<std::pair<Point_2, FT> > points_2;
  points_2.push_back(std::make_pair(Point_2(1.0, 0.0),  1.0));
  points_2.push_back(std::make_pair(Point_2(2.0, 2.0),  2.0));
  points_2.push_back(std::make_pair(Point_2(3.0, 5.0), -2.0));

  Point_2 c2 = CGAL::barycenter(points_2.begin(), points_2.end());
  std::cout << c2 << std::endl;

  // barycenter of 3D weighted points
  std::list<std::pair<Point_3, FT> > points_3;
  points_3.push_back(std::make_pair(Point_3(1.0, 0.0, 0.5),  1.0));
  points_3.push_back(std::make_pair(Point_3(2.0, 2.0, 1.2),  2.0));
  points_3.push_back(std::make_pair(Point_3(3.0, 5.0, 4.5), -5.0));

  Point_3 c3 = CGAL::barycenter(points_3.begin(), points_3.end());
  std::cout << c3 << std::endl;

  return 0;
}
```

File: examples/Principal_component_analysis/barycenter.cpp

69.2.4 Least Squares Fitting of a Line to a 2D Point Set

In the following example we use a STL container of 2D points, and compute the best fitting line in the least squares sense.

```
#include <CGAL/Cartesian.h>
#include <CGAL/linear_least_squares_fitting_2.h>
#include <list>

typedef double                   FT;
```

```
typedef CGAL::Cartesian<FT>  K;
typedef K::Line_2            Line;
typedef K::Point_2           Point;

int main()
{
  std::list<Point> points;
  points.push_back(Point(1.0,2.0));
  points.push_back(Point(3.0,4.0));
  points.push_back(Point(5.0,6.0));

  // fit line
  Line line;

   linear_least_squares_fitting_2(points.begin(),points.end(),line,CGAL::Dimension_tag<0>

  return 0;
}
```

File: examples/Principal_component_analysis/linear_least_squares_fitting_points_2.cpp

69.2.5 Least Squares Fitting of a Line and a Plane to a 3D Triangle Set

In the following example we use a STL container of 3D triangles, and compute the best fitting line and plane in the least squares sense. Note that we can fit either the whole triangles, the triangle edges or the triangle vertices.

```
#include <CGAL/Cartesian.h>
#include <CGAL/linear_least_squares_fitting_3.h>
#include <list>

typedef double               FT;
typedef CGAL::Cartesian<FT>  K;
typedef K::Line_3            Line;
typedef K::Plane_3           Plane;
typedef K::Point_3           Point;
typedef K::Triangle_3        Triangle;

int main(void)
{
  std::list<Triangle> triangles;
  Point a(1.0,2.0,3.0);
  Point b(4.0,0.0,6.0);
  Point c(7.0,8.0,9.0);
  Point d(8.0,7.0,6.0);
  Point e(5.0,3.0,4.0);
  triangles.push_back(Triangle(a,b,c));
  triangles.push_back(Triangle(a,b,d));
  triangles.push_back(Triangle(d,e,c));

  Line line;
  Plane plane;

```

```
  // fit plane to whole triangles

   linear_least_squares_fitting_3(triangles.begin(),triangles.end(),plane,CGAL::Dimension

  // fit line to triangle vertices
  linear_least_squares_fitting_3(triangles.begin(),triangles.end(),line,
   CGAL::Dimension_tag<0>());

  return 0;
}
```

File: examples/Principal_component_analysis/linear_least_squares_fitting_triangles_3.cpp

Principal Component Analysis Reference Manual

Pierre Alliez, Sylvain Pion and Ankit Gupta

This CGAL component provides functions to compute global information about the shape of a set of 2D or 3D objects. It provides the computation of axis-aligned bounding boxes for point sets, and barycenters of weighted point sets. In addition, it provides computation of centroids (center of mass) and linear least squares fitting for point sets as well as for sets of other bounded objects. More specifically, it is possible to fit 2D lines to 2D segments, circles, disks, iso-rectangles and triangles, as well as to fit 3D lines or 3D planes to 3D segments, triangles, iso-cuboids, tetrahedra, spheres and balls. The interface to these functions takes an iterator range of objects.

69.3 Classified Reference Pages

Functions

69.4 Alphabetical List of Reference Pages

Function

CGAL::bounding_box

Definition

The function *bounding_box* computes the axis-aligned bounding box of a set of 2D or 3D points. The bounding box is returned either as an iso rectangle in 2D or as an iso cuboid in 3D, the type being deduced automatically from the value type of the iterator range.

#include <CGAL/bounding_box.h>

There is a set of overloaded *bounding_box* functions for 2D and 3D points. The user can also optionally pass an explicit kernel, in case the default, based on *Kernel_traits* is not sufficient. The dimension is also deduced automatically.

template < typename InputIterator >
K::Iso_rectangle_2 *bounding_box(InputIterator first, InputIterator beyond)*

computes the bounding box of a non-empty set of 2D points. *K* is *Kernel_traits<std::iterator_traits<InputIterator> ::value_type>::Kernel*. The value type must be *K::Point_2*.
Precondition: first != beyond.

template < typename InputIterator, typename K >
K::Iso_rectangle_2 *bounding_box(InputIterator first, InputIterator beyond, K k)*

computes the bounding box of a non-empty set of 2D points. The value type must be *K::Point_2*.
Precondition: first != beyond.

template < typename InputIterator >
K::Iso_cuboid_3 *bounding_box(InputIterator first, InputIterator beyond)*

computes the bounding box of a non-empty set of 3D points. *K* is *Kernel_traits<std::iterator_traits<InputIterator> ::value_type>::Kernel*. The value type must be *K::Point_3*.
Precondition: first != beyond.

template < typename InputIterator, typename K >
K::Iso_cuboid_3 *bounding_box(InputIterator first, InputIterator beyond, K k)*

computes the bounding box of a non-empty set of 3D points. The value type must be *K::Point_3*.
Precondition: first != beyond.

CGAL::centroid

Function

Definition

The function *centroid* computes the (uniform) center of mass of a set of 2D or 3D bounded objects. In 2D these objects include points, segments, triangles, iso rectangles, circles and disks. In 3D these objects include points, segments, triangles, iso cuboids, spheres, balls and tetrahedra.

#include <CGAL/centroid.h>

There is a set of overloaded *centroid* functions for 2D and 3D objects. The user can also optionally pass an explicit kernel, in case the default based on *Kernel_traits* is not sufficient. The default dimension tag is deduced automatically, although the user can pass a *tag* specifying the dimension of the objects to be considered for the centroid computation. For example, the default dimension of a tetrahedron is 3, but specifying a dimension 0 computes the centroid of the tetrahedron vertices (3D points), specifying a dimension 1 computes the centroid of the tetrahedron edges (3D segments) and specifying a dimension 2 computes the centroid of the tetrahedron facets (3D triangles).

template < typename InputIterator, typename Tag >
K::Point_2 *centroid(InputIterator first, InputIterator beyond, Tag t)*

computes the centroid of a non-empty set of 2D objects. The tag is used to specify the dimension to be considered from the objects. *K* is *Kernel_traits<std::iterator_traits<InputIterator>::value_type>::Kernel*. The value type must be either *K::Point_2*, *K::Segment_2*, *K::Triangle_2*, *K::Rectangle_2* or *K::Circle_2*. To fit a set of disks the user must call the function with value type *K::Circle_2* and with dimension tag of 2. The tag must range between *CGAL::Dimension_tag<0>* and *CGAL::Dimension_tag<2>*.
Precondition: first != beyond.

template < typename InputIterator, typename K, typename Tag >
K::Point_2 *centroid(InputIterator first, InputIterator beyond, K k, Tag t)*

computes the centroid of a non-empty set of 2D objects. The tag is used to specify the dimension to be considered from the objects. *K* is *Kernel_traits<std::iterator_traits<InputIterator>::value_type>::Kernel*. The value type must be either *K::Point_2*, *K::Segment_2*, *K::Triangle_2*, *K::Rectangle_2* or *K::Circle_2*. To fit a set of disks the user must call the function with value type *K::Circle_2* and with dimension tag of 2. The tag must range between *CGAL::Dimension_tag<0>* and *CGAL::Dimension_tag<2>*.
Precondition: first != beyond.

template < typename InputIterator, typename Tag >

K::Point_3 *centroid(InputIterator first, InputIterator beyond, Tag t)*

computes the centroid of a non-empty set of 3D objects. The tag is used to specify the dimension to be considered from the objects. *K* is *Kernel_traits<std::iterator_traits< InputIterator>::value_type>::Kernel*. The value type must be either *K::Point_3*, *K::Segment_3*, *Triangle_3*, *Cuboid_3*, *Sphere_3* or *Tetrahedron_3*. To fit a set of balls the user must call the function with value type *K::Sphere_3* and with dimension tag of 3. The tag must range between *CGAL::Dimension_tag<0>* and *CGAL::Dimension_tag<3>*.
Precondition: first != beyond.

template < typename InputIterator, typename K, typename Tag >
K::Point_3 *centroid(InputIterator first, InputIterator beyond, K k, Tag t)*

computes the centroid of a non-empty set of 3D objects. The tag is used to specify the dimension to be considered from the objects. *K* is *Kernel_traits<std::iterator_traits< InputIterator>::value_type>::Kernel*. The value type must be either *K::Point_3*, *K::Segment_3*, *Triangle_3*, *Cuboid_3*, *Sphere_3* or *Tetrahedron_3*. To fit a set of balls the user must call the function with value type *K::Sphere_3* and with dimension tag of 3. The tag must range between *CGAL::Dimension_tag<0>* and *CGAL::Dimension_tag<3>*.
Precondition: first != beyond.

See Also

CGAL::barycenter . . . . page 3971

CGAL::barycenter

Function

Definition

The function *barycenter* computes the barycenter (weighted center of mass) of a set of 2D or 3D weighted points. The weight associated to each point is specified using a *std::pair* storing the point and its weight.

#include <CGAL/barycenter.h>

There is a set of overloaded *barycenter* functions for 2D and 3D weighted points. The user can also optionally pass an explicit kernel, in case the default, based on *Kernel_traits* is not sufficient. The dimension is also deduced automatically.

template < typename InputIterator >
K::Point_2 *barycenter(InputIterator first, InputIterator beyond)*

computes the barycenter of a non-empty set of 2D weighted points. *K* is *Kernel_traits<std::iterator_traits<InputIterator> ::value_type::first_type>::Kernel*. The value type must be *std::pair<K::Point_2, K::FT>*.
Precondition: first != beyond, and the sum of the weights is non-zero.

template < typename InputIterator, typename K >
K::Point_2 *barycenter(InputIterator first, InputIterator beyond, K k)*

computes the barycenter of a non-empty set of 2D weighted points. The value type must be *std::pair<K::Point_2, K::FT>*.
Precondition: first != beyond, and the sum of the weights is non-zero.

template < typename InputIterator >
K::Point_3 *barycenter(InputIterator first, InputIterator beyond)*

computes the barycenter of a non-empty set of 3D weighted points. *K* is *Kernel_traits<std::iterator_traits<InputIterator> ::value_type::first_type>::Kernel*. The value type must be *std::pair<K::Point_3, K::FT>*.
Precondition: first != beyond, and the sum of the weights is non-zero.

template < typename InputIterator, typename K >
K::Point_3 *barycenter(InputIterator first, InputIterator beyond, K k)*

computes the barycenter of a non-empty set of 3D weighted points. The value type must be *std::pair<K::Point_3, K::FT>*.
Precondition: first != beyond, and the sum of the weights is non-zero.

See Also

CGAL::centroid . . . . page 3969

CGAL::linear_least_squares_fitting_2

Function

Definition

The function *linear_least_squares_fitting_2* computes the 2D best fitting line (in the least squares sense) of a set of 2D objects such as points, segments, triangles, iso rectangles, circles or disks.

The best fitting line minimizes the sum of squared distances from all points comprising these objects to their orthogonal projections onto the line. It can be shown that this line goes through the centroid of the set. This problem is equivalent to search for the linear sub-space which maximizes the variance of projected points (sum of squared distances to the centroid). Internally we solve this problem by eigen decomposition of the covariance matrix of the whole set. Note that the 2×2 covariance matrix is computed internally in closed form and not by point sampling the objects. Eigenvectors corresponding to large eigenvalues are the directions in which the data has strong component, or equivalently large variance. If one eigenvalue is null the fit is perfect as the sum of squared distance from all points to their projection onto the best line is null. If the two eigenvalues are the same there is no preferable sub-space and all lines going through the centroid share the same fitting property.

#include <CGAL/linear_least_squares_fitting_2.h>

```
template < typename InputIterator, typename K, typename Tag >
typename K::FT      linear_least_squares_fitting_2( InputIterator first,
                                                    InputIterator beyond,
                                                    typename K::Line_2 & line,
                                                    typename K::Point_2 & centroid,
                                                    Tag tag,
                                                    K k)
```

computes the best fitting 2D line of a 2D object set in the range [*first*,*beyond*). The value returned is a fitting quality between 0 and 1, where 0 means that the variance is the same along any line (a horizontal line going through the centroid is output by default), and 1 means that the variance is null orthogonally to the best fitting line (hence the fit is perfect).
Precondition: first != beyond.

The tag *tag* identifies the dimension to be considered from the objects. For point sets it should be 0. For segments it can be 1 or 0 according to whether one wants to fit the whole segment or just their end points. For triangles it can range from 0 to 2 according to whether one wants to fit either the triangle points, the segments or the whole triangles. For rectangles it can range from 0 to 2 according to whether one wants to fit either the corner points, the segments, or the whole rectangles. For circles it can be 1 or 2 according to whether one wants to fit either the circles or the whole discs. For triangles it ranges from 0 to 2 according to whether one wants to fit either the points, the segments or the whole triangles.

The class *K* is the kernel in which the type *InputIterator::value_type* is defined. It can be omitted and deduced automatically from the value type.

Requirements

1. *InputIterator::value_type* is equivalent to *K::Point_2* or *K::Segment_2* or *K::Triangle_2* or *K::Rectangle_2* or *K::Circle_2*.

2. *line* is the best fitting line computed.

3. *centroid* is the centroid computed. This parameter is optional and can be omitted.

4. *tag* is the tag identifying the dimension to be considered from the objects. It should be one of *Dimension_tag<0>*, *Dimension_tag<1>* or *Dimension_tag<2>*. Also, it should not be of dimension greater than the geometry of the object. For example, a *Segment* can not have a *Dimension_tag<2>* tag.

CGAL::linear_least_squares_fitting_3

Function

Definition

Function *linear_least_squares_fitting_3* computes the best fitting 3D line or plane (in the least squares sense) of a set of 3D objects such as points, segments, triangles, spheres, balls, cuboids or tetrahedra.

The best fitting linear sub-space (here line or plane) minimizes the sum of squared distances from all points comprising these objects to their orthogonal projections onto this linear subspace. It can be shown that the best line or plane goes through the centroid of the set. This problem is equivalent to search for the linear sub-space which maximizes the variance of projected points (sum of squared distances to the centroid). Internally we solve this problem by eigen decomposition of the covariance matrix of the whole set. Note that the 3×3 covariance matrix is computed internally in closed form and not by point sampling the objects. Eigenvectors corresponding to large eigenvalues are the directions in which the data has strong component, or equivalently large variance.

The fitting quality property is characterized by the values of the three eigenvalues. When all three values are distinct the best linear subspace is uniquely determined, be it a line or a plane. When all three eigenvalues are equal there is no preferable sub-space and any line or plane going through the centroid share the same fitting property (a horizontal plane or a line along the x axis are returned by default). A best fitting line is uniquely determined as soon as the largest eigenvalue is different from the two others, otherwise all lines contained in the best fitting plane share the same fitting property. A best fitting plane is uniquely determined as soon as the smallest eigenvalue is different from the two others, otherwise all planes going through the best fitting line share the same fitting property.

#include <CGAL/linear_least_squares_fitting_3.h>

```
template < typename InputIterator, typename K, typename Tag >
typename K::FT     linear_least_squares_fitting_3( InputIterator first,
                                                   InputIterator beyond,
                                                   typename K::Line_3& line,
                                                   typename K::Point_3& centroid,
                                                   Tag tag,
                                                   K k)
```

computes the best fitting 3D line of a 3D object set in the range [*first*,*beyond*). The value returned is a fitting quality between 0 and 1, where 0 means that the variance is the same along any line contained within the best fitting plane, and 1 means that the variance is null orthogonally to the best fitting line (hence the fit is perfect).

The tag *tag* identifies the dimension to be considered from the objects. For point sets it should be 0. For segment sets it could be 1 or 0 according to whether one wants to fit the entire segments or just the end points. For triangle sets it can range from 0 to 2 according to whether one wants to fit either the corner points, the segments or the whole triangles. For cuboid sets it can range from 0 to 3 according to whether one wants to fit either the corners, the segments, the faces or the whole solid cuboids. For sphere sets it can be 2 or 3 according to whether one wants to fit either the surface of the spheres or the whole solid balls. For tetrahedron sets it can range from 0 to

3 according to whether one wants to fit either the points, the segments, the surface triangles or the whole solid tetrahedra.

The class *K* is the kernel in which the type *InputIterator::value_type* is defined. It can be omitted and deduced automatically from the value type.

Requirements

1. *InputIterator::value_type* can be either *K::Point_3*, *K::Segment_3*, *Triangle_3*, *Cuboid_3*, *Sphere_3* or *Tetrahedron_3*.
2. *line* is the best fitting line computed.
3. *centroid* is the centroid computed. This parameter is optional and can be omitted.
4. *tag* is the tag identifying the dimension to be considered from the objects. It should range from *Dimension_tag<0>* to *Dimension_tag<3>*. Also, it should not be of a dimension greater nor smaller than the geometry of the object. For example, a *Triangle* can not have a *Dimension_tag<3>* tag. A *Segment* can not have a *Dimension_tag<2>* nor a *Dimension_tag<3>* tag. A *Sphere* can not have a *Dimension_tag<0>* nor a *Dimension_tag<1>* tag.

template < typename InputIterator, typename K, typename Tag >
typename K::FT linear_least_squares_fitting_3(InputIterator first, InputIterator beyond, typename K::Plane_3& plane, typename K::Point_3& centroid, Tag tag, K k)

computes the best fitting 3D plane of a 3D object set in the range [*first*,*beyond*). The value returned is a fitting quality between 0 and 1, where 0 means that the variance is the same along any plane going through the best fitting line, and 1 means that the variance is null orthogonally to the best fitting plane (hence the fit is perfect).

The class *K* is the kernel in which the type *InputIterator::value_type* is defined. It can be omitted and deduced automatically from the value type. The tag *tag* identifies the dimension to be considered from the objects (see above).

Requirements

1. *InputIterator::value_type* can either be *K::Point_3*, *K::Segment_3*, *Triangle_3*, *Cuboid_3*, *Sphere_3* or *Tetrahedron_3*.
2. *plane* is the best fitting plane computed.
3. *centroid* is the centroid computed. This parameter is optional and can be omitted.
4. *tag* is the tag identifying the dimension to be considered from the objects. It should range from *Dimension_tag<0>* to *Dimension_tag<3>*. Also, it should not be of a dimension greater nor smaller than the geometry of the object. For example, a *Triangle* can not have a *Dimension_tag<3>* tag. A *Segment* can not have a *Dimension_tag<2>* nor a *Dimension_tag<3>* tag. A *Sphere* can not have a *Dimension_tag<0>* nor a *Dimension_tag<1>* tag.

Part XV

Interpolation

Chapter 70

2D and Surface Function Interpolation

Julia Flötotto

Contents

This chapter describes CGAL's interpolation package which implements natural neighbor coordinate functions as well as different methods for scattered data interpolation most of which are based on natural neighbor coordinates. The functions for computing natural neighbor coordinates in Euclidean space are described in Section 70.1, the functions concerning the coordinate and neighbor computation on surfaces are discussed in Section 70.2. In Section 70.3, we describe the different interpolation functions.

Scattered data interpolation solves the following problem: given measures of a function on a set of discrete data points, the task is to interpolate this function on an arbitrary query point. More formally, let $\mathcal{P} = \{\mathbf{p_1}, \ldots, \mathbf{p_n}\}$ be a set of n points in $\mathbb{R}^2$ or $\mathbb{R}^3$ and Φ be a scalar function defined on the convex hull of $\mathcal{P}$. We assume that the function values are known at the points of $\mathcal{P}$, i.e. to each $\mathbf{p_i} \in \mathcal{P}$, we associate $z_i = \Phi(\mathbf{p_i})$. Sometimes, the gradient of Φ is also known at $\mathbf{p_i}$. It is denoted $\mathbf{g_i} = \nabla\Phi(\mathbf{p_i})$. The interpolation is carried out for an arbitrary query point $\mathbf{x}$ on the convex hull of $\mathcal{P}$.

70.1 Natural Neighbor Coordinates

70.1.1 Introduction

Natural neighbor interpolation has been introduced by Sibson [Sib81] to interpolate multivariate scattered data. Given a set of data points $\mathcal{P}$, the natural neighbor coordinates associated to $\mathcal{P}$ are defined from the Voronoi diagram of $\mathcal{P}$. When simulating the insertion of a query point $\mathbf{x}$ into the Voronoi diagram of $\mathcal{P}$, the potential Voronoi cell of $\mathbf{x}$ "steals" some parts from the existing cells.

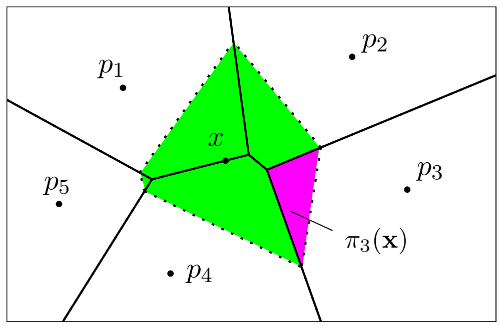


Figure 70.1: $2D$ example: $\mathbf{x}$ has five natural neighbors $\mathbf{p_1},\ldots,\mathbf{p_5}$. The natural neighbor coordinate $\lambda_3(\mathbf{x})$ is the ratio of the area of the pink polygon, $\pi_3(\mathbf{x})$, over the area of the total highlighted zone.

Let $\pi(\mathbf{x})$ denote the volume of the potential Voronoi cell of $\mathbf{x}$ and $\pi_i(\mathbf{x})$ denote the volume of the sub-cell that would be stolen from the cell of $\mathbf{p_i}$ by the cell of $\mathbf{x}$. The natural neighbor coordinate of $\mathbf{x}$ with respect to the data point $\mathbf{p_i} \in \mathcal{P}$ is defined by

$$\lambda_i(\mathbf{x}) = \frac{\pi_i(\mathbf{x})}{\pi(\mathbf{x})}.$$

A two-dimensional example is depicted in Figure 70.1.

Various papers ([Sib80], [Far90], [Pip93], [Bro97], [HS00]) show that the natural neighbor coordinates have the following properties:

1. $\mathbf{x} = \sum_{i=1}^{n} \lambda_i(\mathbf{x})\mathbf{p_i}$ (barycentric coordinate property).
2. For any $i, j \leq n, \lambda_i(\mathbf{p_j}) = \delta_{ij}$, where δ_{ij} is the Kronecker symbol.
3. $\sum_{i=1}^{n} \lambda_i(\mathbf{x}) = 1$ (partition of unity property).

For the case where the query point x is located on the envelope of the convex hull of $\mathcal{P}$, the potential Voronoi cell of x becomes infinite and :

$\pi(\mathbf{x}) = \infty$

$\lambda_i(\mathbf{x}) = 0$ for all data point $\mathbf{p_i}$ of $\mathcal{P}$ except for the two endpoints, let's say $\mathbf{p}$ and $\mathbf{q}$,of the edge where x lies.

The natural neighbor coordinate of $\mathbf{x}$ with respect to these endpoints $\mathbf{p}$ and $\mathbf{q}$ will be :

$\lambda_p(\mathbf{x}) = \frac{\|\mathbf{x}-\mathbf{q}\|}{\|\mathbf{q}-\mathbf{p}\|}$

$\lambda_q(\mathbf{x}) = \frac{\|\mathbf{x}-\mathbf{p}\|}{\|\mathbf{q}-\mathbf{p}\|}$

Furthermore, Piper [Pip93] shows that the coordinate functions are continuous in the convex hull of $\mathcal{P}$ and continuously differentiable except on the data points $\mathcal{P}$.

The interpolation package of CGAL provides functions to compute natural neighbor coordinates for $2D$ and $3D$ points with respect to Voronoi diagrams as well as with respect to power diagrams (only $2D$), i.e. for weighted points. Refer to the reference pages *natural_neighbor_coordinates_2*, *natural_neighbor_coordinates_3* and *regular_neighbor_coordinates_2*.

In addition, the package provides functions to compute natural neighbor coordinates on well sampled point set surfaces. See Section 70.2 and the reference page *surface_neighbor_coordinates_3* for further information.

70.1.2 Implementation

Given a Delaunay triangulation or a Regular triangulation, the vertices in conflict with the query point are determined. The areas $\pi_i(\mathbf{x})$ are computed by triangulating the Voronoi sub-cells. The normalization factor $\pi(\mathbf{x})$ is also returned. If the query point is already located and/or the boundary edges of the conflict zone are already determined, alternative functions allow to avoid the re-computation.

70.1.3 Example for Natural Neighbor Coordinates

The signature of all coordinate computation functions is about the same.

```
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/Delaunay_triangulation_2.h>
#include <CGAL/natural_neighbor_coordinates_2.h>

typedef CGAL::Exact_predicates_inexact_constructions_kernel K;
typedef CGAL::Delaunay_triangulation_2<K>
   Delaunay_triangulation;
typedef std::vector< std::pair< K::Point_2, K::FT  > >

   Point_coordinate_vector;

int main()
{
  Delaunay_triangulation dt;

  for (int y=0 ; y<3 ; y++)
    for (int x=0 ; x<3 ; x++)
      dt.insert(K::Point_2(x,y));

  //coordinate computation
  K::Point_2 p(1.2, 0.7);
  Point_coordinate_vector coords;
  CGAL::Triple<
    std::back_insert_iterator<Point_coordinate_vector>,
    K::FT, bool> result =
    CGAL::natural_neighbor_coordinates_2(dt, p,
                                         std::back_inserter(coords));
  if(!result.third){
    std::cout << "The coordinate computation was not successful."
```

```
                << std::endl;
    std::cout << "The point (" <<p << ") lies outside the convex hull."
                << std::endl;
  }
  K::FT  norm = result.second;
  std::cout << "Coordinate computation successful." << std::endl;
  std::cout << "Normalization factor: " <<norm << std::endl;
  std::cout << "done" << std::endl;
  return 0;
}
```

File: examples/Interpolation/nn_coordinates_2.cpp

70.1.4 Example for Regular Neighbor Coordinates

For regular neighbor coordinates, it is sufficient to replace the name of the function and the type of triangulation passed as parameter. A special traits class is needed.

```
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/Regular_triangulation_2.h>
#include <CGAL/Regular_triangulation_euclidean_traits_2.h>
#include <CGAL/regular_neighbor_coordinates_2.h>

typedef CGAL::Exact_predicates_inexact_constructions_kernel K;

typedef CGAL::Regular_triangulation_euclidean_traits_2<K> Gt;
typedef CGAL::Regular_triangulation_2<Gt>
   Regular_triangulation;
typedef Regular_triangulation::Weighted_point Weighted_point;
typedef std::vector< std::pair< Weighted_point, K::FT  > >

   Point_coordinate_vector;

int main()
{
  Regular_triangulation rt;

  for (int y=0 ; y<3 ; y++)
    for (int x=0 ; x<3 ; x++)
      rt.insert(Weighted_point(K::Point_2(x,y), 0));

  //coordinate computation
  Weighted_point wp(K::Point_2(1.2, 0.7),2);
  Point_coordinate_vector  coords;
  CGAL::Triple<
    std::back_insert_iterator<Point_coordinate_vector>,
    K::FT, bool> result =
    CGAL::regular_neighbor_coordinates_2(rt, wp,
                                         std::back_inserter(coords));
  if(!result.third){
    std::cout << "The coordinate computation was not successful."
                << std::endl;
```

```
    std::cout << "The point (" <<wp.point() << ") lies outside the convex
    hull."
              << std::endl;
  }
  K::FT  norm = result.second;
  std::cout << "Coordinate computation successful." << std::endl;
  std::cout << "Normalization factor: " <<norm << std::endl;

  std::cout << "done" << std::endl;
  return 0;
}
```

File: examples/Interpolation/rn_coordinates_2.cpp

For surface neighbor coordinates, the surface normal at the query point must be provided, see Section 70.2.

70.2 Surface Natural Neighbor Coordinates and Surface Neighbors

This section introduces the functions to compute natural neighbor coordinates and surface neighbors associated to a set of sample points issued from a surface $\mathcal{S}$ and given a query point $\mathbf{x}$ on $\mathcal{S}$. We suppose that $\mathcal{S}$ is a closed and compact surface of $\mathbb{R}^3$, and let $\mathcal{P} = \{\mathbf{p_1}, \ldots, \mathbf{p_n}\}$ be an ε-sample of $\mathcal{S}$ (refer to Amenta and Bern [AB99]). The concepts are based on the definition of Boissonnat and Flötotto [BF02], [Flö03b]. Both references contain a thorough description of the requirements and the mathematical properties.

70.2.1 Introduction

Two observations lead to the definition of surface neighbors and surface neighbor coordinates: First, it is clear that the tangent plane $\mathcal{T}_x$ of the surface $\mathcal{S}$ at the point $\mathbf{x} \in \mathcal{S}$ approximates $\mathcal{S}$ in the neighborhood of $\mathbf{x}$. It has been shown in [BF02] that, if the surface $\mathcal{S}$ is well sampled with respect to the curvature and the local thickness of $\mathcal{S}$, i.e. it is an ε-sample, the intersection of the tangent plane $\mathcal{T}_x$ with the Voronoi cell of $\mathbf{x}$ in the Voronoi diagram of $\mathcal{P} \cup \{\mathbf{x}\}$ has a small diameter. Consequently, inside this Voronoi cell, the tangent plane $\mathcal{T}_x$ is a reasonable approximation of $\mathcal{S}$. Furthermore, the second observation allows to compute this intersection diagram easily: one can show using Pythagoras' Theorem that the intersection of a three-dimensional Voronoi diagram with a plane $\mathcal{H}$ is a two-dimensional power diagram. The points defining the power diagram are the projections of the points in $\mathcal{P}$ onto $\mathcal{H}$, each point weighted with its negative square distance to $\mathcal{H}$. Algorithms for the computation of power diagrams via the dual regular triangulation are well known and for example provided by CGAL in the class *Regular_triangulation_2<Gt, Tds>*.

70.2.2 Implementation

Voronoi Intersection Diagrams

In CGAL, the regular triangulation dual to the intersection of a 3D Voronoi diagram with a plane $\mathcal{H}$ can be computed by instantiating the *Regular_triangulation_2<Gt, Tds>* class with the traits class *Voronoi_intersection_2_traits_3<K>*. This traits class contains a point and a vector as class member which define the plane $\mathcal{H}$. All predicates and constructions used by *Regular_triangulation_2<Gt, Tds>* are replaced by the corresponding operators on three-dimensional points. For example, the power test predicate (which takes three weighted $2D$ points p', q', r' of the regular triangulation and tests the power distance of a fourth point t' with respect to the power circle orthogonal to p, q, r) is replaced by a *Side_of_plane_centered_sphere_2_3* predicate that tests the position of a $3D$ point t with respect to the sphere centered on the plane $\mathcal{H}$ passing through the $3D$ points p, q, r. This approach allows to avoid the explicit constructions of the projected points and the weights which are very prone to rounding errors.

Natural Neighbor Coordinates on Surfaces

The computation of natural neighbor coordinates on surfaces is based upon the computation of regular neighbor coordinates with respect to the regular triangulation that is dual to $\mathrm{Vor}(\mathcal{P}) \cap \mathcal{T}_x$, the intersection of $\mathcal{T}_x$ and the Voronoi diagram of $\mathcal{P}$, via the function *regular_neighbor_coordinates_2*.

Of course, we might introduce all data points $\mathcal{P}$ into this regular triangulation. However, this is not necessary because we are only interested in the cell of $\mathbf{x}$. It is sufficient to guarantee that all surface neighbors of the query point $\mathbf{x}$ are among the input points that are passed as argument to the function. The sample points $\mathcal{P}$ can be filtered for example by distance, e.g. using range search or k-nearest neighbor queries, or with the help of the $3D$ Delaunay triangulation since the surface neighbors are necessarily a subset of the natural neighbors of the query point in this triangulation. CGAL provides a function that encapsulates the filtering based on the $3D$ Delaunay triangulation. For input points filtered by distance, functions are provided that indicate whether or not points that lie outside the input range (i.e. points that are further from $\mathbf{x}$ than the furthest input point) can still influence the result. This allows to iteratively enlarge the set of input points until the range is sufficient to certify the result.

Surface Neighbors

The surface neighbors of the query point are its neighbors in the regular triangulation that is dual to $\mathrm{Vor}(\mathcal{P}) \cap \mathcal{T}_x$, the intersection of $\mathcal{T}_x$ and the Voronoi diagram of $\mathcal{P}$. As for surface neighbor coordinates, this regular triangulation is computed and the same kind of filtering of the data points as well as the certification described above is provided.

70.2.3 Example for Surface Neighbor Coordinates

```
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>

#include <CGAL/point_generators_3.h>
#include <CGAL/algorithm.h>
#include <CGAL/Origin.h>

#include <CGAL/surface_neighbor_coordinates_3.h>

typedef CGAL::Exact_predicates_inexact_constructions_kernel K;
typedef K::FT                           Coord_type;
typedef K::Point_3                      Point_3;
typedef K::Vector_3                     Vector_3;
typedef std::vector< std::pair< Point_3, K::FT  > >
                                        Point_coordinate_vector;
```

```
int main()
{

  int n=100;
  std::vector< Point_3> points;
  points.reserve(n);

  std::cout << "Generate " << n << " random points on a sphere."
            << std::endl;
  CGAL::Random_points_on_sphere_3<Point_3> g(1);
  CGAL::cpp0x::copy_n( g, n, std::back_inserter(points));

  Point_3 p(1, 0,0);
  Vector_3 normal(p-CGAL::ORIGIN);
  std::cout << "Compute surface neighbor coordinates for "
            << p << std::endl;
  Point_coordinate_vector coords;
  CGAL::Triple< std::back_insert_iterator<Point_coordinate_vector>,
    K::FT, bool> result =
    CGAL::surface_neighbor_coordinates_3(points.begin(), points.end(),
                                         p, normal,
                                         std::back_inserter(coords),
                                         K());
  if(!result.third){
    //Undersampling:
    std::cout << "The coordinate computation was not successful."
              << std::endl;
    return 0;
  }
  K::FT norm = result.second;

  std::cout << "Testing the barycentric property " << std::endl;
  Point_3 b(0, 0,0);
  for(std::vector< std::pair< Point_3, Coord_type  > >::const_iterator
        it = coords.begin(); it!=coords.end(); ++it)
    b = b + (it->second/norm)* (it->first - CGAL::ORIGIN);

  std::cout <<"    weighted barycenter: " << b <<std::endl;
  std::cout << "    squared distance: " <<
    CGAL::squared_distance(p,b) <<std::endl;

  std::cout << "done" << std::endl;
  return 0;
}
```

File: examples/Interpolation/surface_neighbor_coordinates_3.cpp

70.3 Interpolation Methods

70.3.1 Introduction

Linear Precision Interpolation

Sibson [Sib81] defines a very simple interpolant that re-produces linear functions exactly. The interpolation of $\Phi(\mathbf{x})$ is given as the linear combination of the neighbors' function values weighted by the coordinates:

$$Z^0(\mathbf{x}) = \sum_i \lambda_i(\mathbf{x}) z_i.$$

Indeed, if $z_i = a + \mathbf{b}^t \mathbf{p_i}$ for all natural neighbors of $\mathbf{x}$, we have

$$Z^0(\mathbf{x}) = \sum_i \lambda_i(\mathbf{x})(a + \mathbf{b}^t \mathbf{p_i}) = a + \mathbf{b}^t \mathbf{x}$$

by the barycentric coordinate property. The first example in Subsection 70.3.3 shows how the function is called.

Sibson's C^1 Continuous Interpolant

In [Sib81], Sibson describes a second interpolation method that relies also on the function gradient $\mathbf{g_i}$ for all $\mathbf{p_i} \in \mathcal{P}$. It is C^1 continuous with gradient $\mathbf{g_i}$ at $\mathbf{p_i}$. Spherical quadrics of the form $\Phi(\mathbf{x}) = a + \mathbf{b}^t\mathbf{x} + \gamma\, \mathbf{x}^t\mathbf{x}$ are reproduced exactly. The proof relies on the barycentric coordinate property of the natural neighbor coordinates and assumes that the gradient of Φ at the data points is known or approximated from the function values as described in [Sib81] (see Section 70.3.2).

Sibson's Z^1 interpolant is a combination of the linear interpolant Z^0 and an interpolant ξ which is the weighted sum of the first degree functions

$$\xi_i(\mathbf{x}) = z_i + \mathbf{g_i}^t(\mathbf{x} - \mathbf{p_i}), \qquad \xi(\mathbf{x}) = \frac{\sum_i \frac{\lambda_i(\mathbf{x})}{\|\mathbf{x}-\mathbf{p_i}\|}\xi_i(\mathbf{x})}{\sum_i \frac{\lambda_i(\mathbf{x})}{\|\mathbf{x}-\mathbf{p_i}\|}}.$$

Sibson observed that the combination of Z^0 and ξ reconstructs exactly a spherical quadric if they are mixed as follows:

$$Z^1(\mathbf{x}) = \frac{\alpha(\mathbf{x})Z^0(\mathbf{x}) + \beta(\mathbf{x})\xi(\mathbf{x})}{\alpha(\mathbf{x}) + \beta(\mathbf{x})} \text{ where } \alpha(\mathbf{x}) = \frac{\sum_i \lambda_i(\mathbf{x})\frac{\|\mathbf{x}-\mathbf{p_i}\|^2}{f(\|\mathbf{x}-\mathbf{p_i}\|)}}{\sum_i \frac{\lambda_i(\mathbf{x})}{f(\|\mathbf{x}-\mathbf{p_i}\|)}} \text{ and } \beta(\mathbf{x}) = \sum_i \lambda_i(\mathbf{x})\|\mathbf{x}-\mathbf{p_i}\|^2,$$

where in Sibson's original work, $f(\|\mathbf{x}-\mathbf{p_i}\|) = \|\mathbf{x}-\mathbf{p_i}\|$.

CGAL contains a second implementation with $f(\|\mathbf{x}-\mathbf{p_i}\|) = \|\mathbf{x}-\mathbf{p_i}\|^2$ which is less demanding on the number type because it avoids the square-root computation needed to compute the distance $\|\mathbf{x}-\mathbf{p_i}\|$. The theoretical guarantees are the same (see [Flö03b]). Simply, the smaller the slope of f around $f(0)$, the faster the interpolant approaches ξ_i as $\mathbf{x} \rightarrow \mathbf{p_i}$.

Farin's C^1 Continuous Interpolant

Farin [Far90] extended Sibson's work and realizes a C^1 continuous interpolant by embedding natural neighbor coordinates in the Bernstein-Bézier representation of a cubic simplex. If the gradient of Φ at the data points is known, this interpolant reproduces quadratic functions exactly. The function gradient can be approximated from the function values by Sibson's method [Sib81] (see Section 70.3.2) which is exact only for spherical quadrics.

Quadratic Precision Interpolants

Knowing the gradient $\mathbf{g_i}$ for all $\mathbf{p_i} \in \mathcal{P}$, we formulate a very simple interpolant that reproduces exactly quadratic functions. This interpolant is not C^1 continuous in general. It is defined as follows:

$$I^1(\mathbf{x}) = \sum_i \lambda_i(\mathbf{x})(z_i + \frac{1}{2}\mathbf{g_i}^t(\mathbf{x} - \mathbf{p_i}))$$

70.3.2 Gradient Fitting

Sibson describes a method to approximate the gradient of the function f from the function values on the data sites. For the data point $\mathbf{p_i}$, we determine

$$\mathbf{g_i} = \min_{\mathbf{g}} \sum_j \frac{\lambda_j(\mathbf{p_i})}{\|\mathbf{p_i} - \mathbf{p_j}\|^2} \left(z_j - (z_i + \mathbf{g}^t(\mathbf{p_j} - \mathbf{p_i}))\right),$$

where $\lambda_j(\mathbf{p_i})$ is the natural neighbor coordinate of $\mathbf{p_i}$ with respect to $\mathbf{p_i}$ associated to $\mathcal{P} \setminus \{\mathbf{p_i}\}$. For spherical quadrics, the result is exact.

CGAL provides functions to approximate the gradients of all data points that are inside the convex hull. There is one function for each type of natural neighbor coordinate (i.e. *natural_neighbor_coordinates_2*, *regular_neighbor_coordinates_2*).

70.3.3 Example for Linear Interpolation

```
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/Delaunay_triangulation_2.h>

#include <CGAL/Interpolation_traits_2.h>
#include <CGAL/natural_neighbor_coordinates_2.h>
#include <CGAL/interpolation_functions.h>

typedef CGAL::Exact_predicates_inexact_constructions_kernel K;
typedef CGAL::Delaunay_triangulation_2<K>
   Delaunay_triangulation;
typedef CGAL::Interpolation_traits_2<K>               Traits;
typedef K::FT                                         Coord_type;
typedef K::Point_2                                    Point;

int main()
{
  Delaunay_triangulation T;
  std::map<Point, Coord_type, K::Less_xy_2> function_values;
  typedef CGAL::Data_access< std::map<Point, Coord_type, K::Less_xy_2 > >
                                                    Value_access;

  Coord_type a(0.25), bx(1.3), by(-0.7);

  for (int y=0 ; y<3 ; y++)
    for (int x=0 ; x<3 ; x++){
      K::Point_2 p(x,y);
```

```
      T.insert(p);
      function_values.insert(std::make_pair(p,a + bx* x+ by*y));
    }
  //coordinate computation
  K::Point_2 p(1.3,0.34);
  std::vector< std::pair< Point, Coord_type > > coords;
  Coord_type norm =
    CGAL::natural_neighbor_coordinates_2
    (T, p,std::back_inserter(coords)).second;

  Coord_type res =  CGAL::linear_interpolation(coords.begin(),
   coords.end(),
                                              norm,

   Value_access(function_values));

  std::cout << "   Tested interpolation on " << p << " interpolation: "
            << res << " exact: " << a + bx* p.x()+ by* p.y()<< std::endl;

  std::cout << "done" << std::endl;
  return 0;
}
```

File: examples/Interpolation/linear_interpolation_2.cpp

70.3.4 Example for Sibson's C^1 Interpolation Scheme with Gradient Estimation

```
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/Delaunay_triangulation_2.h>

#include <CGAL/natural_neighbor_coordinates_2.h>
#include <CGAL/Interpolation_gradient_fitting_traits_2.h>
#include <CGAL/sibson_gradient_fitting.h>
#include <CGAL/interpolation_functions.h>

typedef CGAL::Exact_predicates_inexact_constructions_kernel K;
typedef CGAL::Delaunay_triangulation_2<K>
   Delaunay_triangulation;
typedef CGAL::Interpolation_gradient_fitting_traits_2<K> Traits;

typedef K::FT                                             Coord_type;
typedef K::Point_2                                        Point;
typedef std::map<Point, Coord_type, K::Less_xy_2>         Point_value_map
   ;
typedef std::map<Point, K::Vector_2 , K::Less_xy_2 >
   Point_vector_map;

int main()
{
  Delaunay_triangulation T;

  Point_value_map function_values;
```

```
  Point_vector_map function_gradients;

  //parameters for spherical function:
  Coord_type a(0.25), bx(1.3), by(-0.7), c(0.2);
  for (int y=0 ; y<4 ; y++)
    for (int x=0 ; x<4 ; x++){
      K::Point_2 p(x,y);
      T.insert(p);
      function_values.insert(std::make_pair(p,a + bx* x+ by*y +
   c*(x*x+y*y)));
    }
  sibson_gradient_fitting_nn_2(T,std::inserter(function_gradients,

   function_gradients.begin()),
                             CGAL::Data_access<Point_value_map>
                             (function_values),
                             Traits());


  //coordiante computation
  K::Point_2 p(1.6,1.4);
  std::vector< std::pair< Point, Coord_type > > coords;
  Coord_type norm =
    CGAL::natural_neighbor_coordinates_2(T, p,std::back_inserter
                                          (coords)).second;


  //Sibson interpolant: version without sqrt:
  std::pair<Coord_type, bool> res =
    CGAL::sibson_c1_interpolation_square
    (coords.begin(),
     coords.end(),norm,p,
     CGAL::Data_access<Point_value_map>(function_values),
     CGAL::Data_access<Point_vector_map>(function_gradients),
     Traits());
  if(res.second)
    std::cout << "   Tested interpolation on " << p
              << " interpolation: " << res.first << " exact: "
              << a + bx * p.x()+ by * p.y()+ c*(p.x()*p.x()+p.y()*p.y())
              << std::endl;
  else
    std::cout << "C^1 Interpolation not successful." << std::endl
              << " not all function_gradients are provided."  <<
   std::endl
              << " You may resort to linear interpolation." << std::endl;

  std::cout << "done" << std::endl;
  return 0;
}
```

File: examples/Interpolation/sibson_interpolation_2.cpp

An additional example in the distribution compares numerically the errors of the different interpolation functions with respect to a known function. It is distributed in the examples directory.

2D and Surface Function Interpolation Reference Manual

Julia Flötotto

Scattered data interpolation solves the following problem: given measures of a function on a set of discrete data points, the task is to interpolate this function on an arbitrary query point.

If the function is a linear function and given barycentric coordinates that allow to express the query point as the convex combination of some data points, the function can be exactly interpolated. If the function gradients are known, we can exactly interpolate quadratic functions given barycentric coordinates. Any further properties of these interpolation functions depend on the properties of the barycentric coordinates. They are provided in this package under the name *linear_interpolation* and *quadratic_interpolation*.

Natural neighbor interpolation

Natural neighbor coordinates are defined by Sibson in 1980 and are based on the Voronoi diagram of the data points. Interpolation methods based on natural neighbor coordinates are particularly interesting because they adapt easily to non-uniform and highly anisotropic data. This package contains Sibson's C^1 continuous interpolation method which interpolates exactly spherical quadrics (of the form $\Phi(\mathbf{x}) = a + \mathbf{b}^t\mathbf{x} + \gamma\,\mathbf{x}^t\mathbf{x}$) and Farin's C^1 continuous interpolation method based on Bernstein-Bézier techniques and interpolating exactly quadratic functions – assuming that the function gradient is known. In addition, Sibson defines a method to approximate the function gradients for data points that are in the interior of the convex hull. This method is exact for spherical quadrics.

This CGAL package implements Sibson's and Farin's interpolation functions as well as Sibson's function gradient fitting method. Furthermore, it provides functions to compute the natural neighbor coordinates with respect to a two-dimensional Voronoi diagram (i. e., from the Delaunay triangulation of the data points) and to a two-dimensional power diagram for weighted points (i. e., from their regular triangulation). Natural neighbor coordinates on closed and well-sampled surfaces can also be computed if the normal to the surface at the query point is known. The latter coordinates are only approximately barycentric, see [BF02].

For a more thorough introduction see the user manual.

70.4 Classified Reference Pages

Concepts

Interpolation Functions

Natural neighbor coordinate computation

Surface neighbor and surface neighbor coordinate computation

70.5 Alphabetical List of Reference Pages

CGAL::linear_interpolation

Function

Definition

The function *linear_interpolation* computes the weighted sum of the function values which must be provided via a functor.

#include <CGAL/interpolation_functions.h>

template < class ForwardIterator, class Functor>
typename Functor::result_type::first_type

linear_interpolation(ForwardIterator first,
ForwardIterator beyond,
typename std::iterator_traits<ForwardIterator>::value_type::second_type norm,
Functor function_values)

ForwardIterator::value_type is a pair associating a point to a (non-normalized) barycentric coordinate. *norm* is the normalization factor. Given a point, the functor *function_values* allows to access a pair of a function value and a Boolean. The Boolean indicates whether the function value could be retrieved correctly. This function generates the interpolated function value as the weighted sum of the values corresponding to each point of the point/coordinate pairs in the range $[first, beyond)$.
Precondition: $norm \neq 0$. *function_value(p).second* == *true* for all points p of the point/coordinate pairs in the range $[first, beyond)$.

Requirements

1. *ForwardIterator::value_type* is a pair of point/coordinate value, thus *ForwardIterator::value_type::first_type* is equivalent to a point and *ForwardIterator::value_type::second_type* is a field number type.
2. *Functor::argument_type* must be equivalent to *ForwardIterator::value_type::first_type* and *Functor::result_type* is a pair of the function value type and a Boolean value. The function value type must provide a multiplication and addition operation with the field number type *ForwardIterator::value_type::second_type* and a constructor with argument 0. A model of the functor is provided by the struct *Data_access*. It must be instantiated accordingly with an associative container (e.g. STL *std::map*) having the point type as *key_type* and the function value type as *mapped_type*.

See Also

CGAL::Data_access<Map> . . . . page 4000
CGAL::natural_neighbor_coordinates_2 . . . . page 4004
CGAL::regular_neighbor_coordinates_2 . . . . page 4006
CGAL::surface_neighbor_coordinates_3 . . . . page 4016

Function

CGAL::sibson_c1_interpolation

Definition

The function *sibson_c1_interpolation* interpolates the function values and the gradients that are provided by functors following the method described in [Sib81].

Parameters

The template parameter *Traits* is to be instantiated with a model of *InterpolationTraits*. *ForwardIterator::value_type* is a pair associating a point to a (non-normalized) barycentric coordinate. *norm* is the normalization factor. The range [*first*,*beyond*) contains the barycentric coordinates for the query point *p*. The functor *function_value* allows to access the value of the interpolated function given a point. *function_gradient* allows to access the function gradient given a point.

#include <CGAL/interpolation_functions.h>

template < class ForwardIterator, class Functor, class GradFunctor, class Traits>
std::pair< typename Functor::result_type, bool>

sibson_c1_interpolation(ForwardIterator first,
ForwardIterator beyond,
typename std::iterator_traits<ForwardIterator>::value_type::second_type norm,
typename std::iterator_traits<ForwardIterator>::value_type:: first_type p,
Functor function_value,
GradFunctor function_gradient,
Traits traits)

This function generates the interpolated function value at the point *p* using Sibson's Z^1 interpolant [Sib81].
If the functor *function_gradient* cannot supply the gradient of a point, the function returns a pair where the Boolean is set to *false*. If the interpolation was successful, the pair contains the interpolated function value as first and *true* as second value.
Precondition: *norm* $\neq 0$. *function_value(p).second* == *true* for all points *p* of the point/coordinate pairs in the range [*first*, *beyond*).

Requirements

1. *Traits* is a model of the concept *InterpolationTraits*.
2. *ForwardIterator::value_type* is a point/coordinate pair. Precisely *ForwardIterator::value_type::first_type* is equivalent to *Traits::Point_d* and *ForwardIterator::value_type::second_type* is equivalent to *Traits::FT*.

3. *Functor::argument_type* must be equivalent to *Traits::Point_d* and *Functor::result_type* is a pair of the function value type and a Boolean. The function value type must provide a multiplication and addition operation with the type *Traits::FT* as well as a constructor with argument 0.

4. *GradFunctor::argument_type* must be equivalent to *Traits::Point_d* and *Functor::result_type* is a pair of the function's gradient type and a Boolean. The function gradient type must provide a multiplication operation with *Traits::Vector_d*.

5. A model of the functor types *Functor* (resp. *GradFunctor*) is provided by the struct *Data_access*. It must be instantiated accordingly with an associative container (e.g. STL *std::map*) having the point type as *key_type* and the function value type (resp. function gradient type) as *mapped_type*.

6. The number type *FT* provided by *Traits* must support the square root operation *sqrt()*.

template < class ForwardIterator, class Functor, class GradFunctor, class Traits>
typename Functor::result_type

sibson_c1_interpolation_square(ForwardIterator first,
ForwardIterator beyond,
typename std::iterator_traits<ForwardIterator> ::value_type::second_type norm,
typename std::iterator_traits<ForwardIterator> ::value_type::first_type p,
Functor function_value,
GradFunctor function_gradient,
Traits traits)

The same as above except that no square root operation is needed for *FT*.

See Also

InterpolationTraits . . . page 4001
GradientFittingTraits . . . page 4010
CGAL::Data_access<Map> . . . page 4000
CGAL::sibson_gradient_fitting . . . page 4008
CGAL::linear_interpolation . . . page 3993
CGAL::Interpolation_traits_2<K> . . . page 4003
CGAL::Interpolation_gradient_fitting_traits_2<K> . . . page 4012
CGAL::natural_neighbor_coordinates_2 . . . page 4004
CGAL::regular_neighbor_coordinates_2 . . . page 4006
CGAL::surface_neighbor_coordinates_3 . . . page 4016

Function

CGAL::farin_c1_interpolation

Definition

The function *farin_c1_interpolation* interpolates the function values and the gradients that are provided by functors using the method described in [Far90].

#include <CGAL/interpolation_functions.h>

Parameters

RandomAccessIterator::value_type is a pair associating a point to a (non-normalized) barycentric coordinate. See *sibson_c1_interpolation* for the other parameters.

template < class RandomAccessIterator, class Functor, class GradFunctor, class Traits>
typename Functor::result_type

farin_c1_interpolation(RandomAccessIterator first,
RandomAccessIterator beyond,
typename std::iterator_traits<RandomAccessIterator>::value_type::second_type norm,
typename std::iterator_traits<ForwardIterator>::value_type::first_type p,
Functor function_value,
GradFunctor function_gradient,
Traits traits)

generates the interpolated function value computed by Farin's interpolant [Far90]. See also *sibson_c1_interpolation*.
Precondition: $norm \neq 0$. *function_value(p).second == true* for all points p of the point/coordinate pairs in the range $[first, beyond)$.
Precondition: The range $[first, beyond)$ contains either one or more than three elements.

Requirements

Same requirements as for *sibson_c1_interpolation* only the iterator must provide random access and *Traits::FT* does not need to provide the square root operation.

See Also

CGAL::Data_access<Map> . . . . . . page 4000
CGAL::linear_interpolation . . . . . . page 3993
CGAL::sibson_c1_interpolation . . . . . . page 3994
CGAL::sibson_gradient_fitting . . . . . . page 4008
CGAL::Interpolation_traits_2<K> . . . . . . page 4003

Function

CGAL::quadratic_interpolation

Definition

The function *quadratic_interpolation* interpolates the function values and first degree functions defined from the function gradients. Both, function values and gradients, must be provided by functors.

#include <CGAL/interpolation_functions.h>

Parameters

See *sibson_c1_interpolation*.

template < class ForwardIterator, class Functor, class GradFunctor, class Traits>
typename Functor::result_type

quadratic_interpolation(ForwardIterator first,
ForwardIterator beyond,
typename std::iterator_traits<ForwardIterator>:: value_type::second_type norm,
typename std::iterator_traits<ForwardIterator>::value_type:: first_type p,
Functor function_value,
GradFunctor function_gradient,
Traits traits)

This function generates the interpolated function value as the weighted sum of the values plus a linear term in the gradient for each point of the point/coordinate pairs in the range [*first*, *beyond*). See also *sibson_c1_interpolation*.
Precondition: *norm* $\neq 0$ *function_value(p).second == true* for all points *p* of the point/coordinate pairs in the range [*first*, *beyond*).

Requirements

Same requirements as for *sibson_c1_interpolation* only that *Traits::FT* does not need to provide the square root operation.

See Also

InterpolationTraits . . . . page 4001
GradientFittingTraits . . . . page 4010
CGAL::Data_access<Map> . . . . page 4000
CGAL::sibson_gradient_fitting . . . . page 4008
CGAL::linear_interpolation . . . . page 3993
CGAL::Interpolation_traits_2<K> . . . . page 4003
CGAL::Interpolation_gradient_fitting_traits_2<K> . . . . page 4012

CGAL::Data_access<Map>

Definition

The struct *Data_access<Map>* implements a functor that allows to retrieve data from an associative container. The functor keeps a reference to the container. Given an instance of the container's key type, it returns a pair of the container's value type and a Boolean indicating whether the retrieval was successful.

This class can be used to provide the values and gradients of the interpolation functions.

#include <CGAL/interpolation_functions.h>

Parameters

The class *Data_access<Map>* has the container type *Map* as template parameter.

Types

typedef Map::mapped_type

Data_type;

typedef Map::key_type

Key_type;

Creation

Data_access<Map> data_access(Map map); — Introduces a *Data_access* to the container *map*.

std::pair< Data_type, bool>

data_access(Key_type p)

If there is an entry for *p* in the container *map*, then the pair of *map.find(p)* and *true* is returned. Otherwise, the Boolean value of the pair is *false*.

InterpolationTraits

Concept

Definition

Most interpolation functions are parameterized by a traits class that defines the primitives used in the interpolation algorithms. The concept InterpolationTraits defines this common set of requirements.

Types

InterpolationTraits:: FT	The number type must follow the model *FieldNumberType*.
InterpolationTraits:: Point_d	The point type on which the function is defined and interpolated.
InterpolationTraits:: Vector_d	The corresponding vector type.
InterpolationTraits:: Construct_vector_d	A constructor object for *Vector_d*. Provides : *Vector_d operator() (Point_d a, Point_d b)* which produces the vector *b - a* and *Vector_d operator() (Null_vector NULL_VECTOR)* which introduces the null vector.
InterpolationTraits:: Construct_scaled_vector_d	Constructor object for *Vector_d*. Provides : *Vector_d operator() (Vector_d v,FT scale)* which produces the vector *v* scaled by a factor *scale*.
InterpolationTraits:: Compute_squared_distance_d	Constructor object for *FT*. Provides the operator: *FT operator() (Point_d a, Point_d b)* returning the squared distance between *a* and *b*.
InterpolationTraits traits;	default constructor.

Construction objects

The following functions that create instances of the above constructor object types must exist.

Construct_vector_d *traits.construct_vector_d_object()*

Construct_scaled_vector_d

traits.construct_scaled_vector_d_object()

Compute_squared_distance_d

traits.compute_squared_distance_d_object()

Has Models

CGAL::Interpolation_traits_2<K> . . . . page 4003
CGAL::Interpolation_gradient_fitting_traits_2<K> . . . . page 4012

See Also

GradientFittingTraits . . . . page 4010
CGAL::sibson_c1_interpolation . . . . page 3994
CGAL::sibson_gradient_fitting . . . . page 4008
CGAL::farin_c1_interpolation . . . . page 3996
CGAL::quadratic_interpolation . . . . page 3998

CGAL::Interpolation_traits_2<K>

Class

Definition

Interpolation_traits_2<K> is a model for the concept *InterpolationTraits* and can be used to instantiate the geometric traits class of interpolation methods applied on a bivariate function over a two-dimensional domain. The traits class is templated by a kernel class *K*.

#include <CGAL/Interpolation_traits_2.h>

Is Model for the Concepts

InterpolationTraits . . . . page 4001

Types

typedef K::FT	*FT;*
typedef K::Point_2	*Point_d;*
typedef K::Vector_2	*Vector_d;*
typedef K::Construct_vector_2	
	Construct_vector_d;
typedef K::Construct_scaled_vector_2	
	Construct_scaled_vector_d;
typedef K::Compute_squared_distance_2	
	Compute_squared_distance_d;

Operations

Construct_scaled_vector_d	
	traits.construct_scaled_vector_d_object() const
Construct_vector_d	*traits.construct_vector_d_object() const*
Compute_squared_distance_d	
	traits.compute_squared_distance_d_object() const

See Also

InterpolationTraits . . . . page 4001
GradientFittingTraits . . . . page 4010
CGAL::Interpolation_gradient_fitting_traits_2<K> . . . . page 4012

Function

CGAL::natural_neighbor_coordinates_2

Definition

The function *natural_neighbor_coordinates_2* computes natural neighbor coordinates, also called Sibson's coordinates, for 2D points provided a two-dimensional triangulation and a query point in the convex hull of the vertices of the triangulation.

#include <CGAL/natural_neighbor_coordinates_2.h>

template < class Dt, class OutputIterator >
CGAL::Triple< OutputIterator, typename Dt::Geom_traits::FT, bool >

natural_neighbor_coordinates_2(Dt dt,
typename Dt::Geom_traits::Point_2 p,
OutputIterator out,
typename Dt::Face_handle start = typename Dt::Face_handle())

computes the natural neighbor coordinates for *p* with respect to the points in the two-dimensional Delaunay triangulation *dt*. The template class *Dt* should be of type *Delaunay_triangulation_2<Traits, Tds>*. The value type of the *OutputIterator* is a pair of *Dt::Point_2* and the coordinate value of type *Dt::Geom_traits::FT*. The sequence of point/coordinate pairs that is computed by the function is placed starting at *out*. The function returns a triple with an iterator that is placed past-the-end of the resulting sequence of point/coordinate pairs, the normalization factor of the coordinates and a Boolean value which is set to true iff the coordinate computation was successful.

template <class Dt, class OutputIterator, class EdgeIterator >
CGAL::Triple< OutputIterator, typename Dt::Geom_traits::FT, bool >

natural_neighbor_coordinates_2(Dt dt,
typename Dt::Geom_traits::Point_2 p,
OutputIterator out,
EdgeIterator hole_begin,
EdgeIterator hole_end)

The same as above. *hole_begin* and *hole_end* determines the iterator range over the boundary edges of the conflict zone of *p* in the triangulation. It is the result of the function *T.get_boundary_of_conflicts(p,std::back_inserter(hole), start)*, see *Delaunay_triangulation_2<Traits, Tds>*.

template <class Dt, class OutputIterator>
CGAL::Triple< OutputIterator, typename Dt::Geom_traits::FT, bool >

natural_neighbor_coordinates_2(Dt dt,

typename Dt::Vertex_handle vh,
OutputIterator out)

This function computes the natural neighbor coordinates of the point *vh->point()* with respect to the vertices of *dt* excluding *vh->point()*. The same as above for the remaining parameters.

Requirements

1. *Dt* are equivalent to the class *Delaunay_triangulation_2<Traits, Tds>*.
2. The traits class *Traits* of *Dt* is a model of the concept *DelaunayTriangulationTraits_2*. Only the following members of this traits class are used:
   - *Construct_circumcenter_2*
   - *FT*
   - *Point_2*
   - *construct_circumcenter_2_object*

   Additionally, *Traits* must meet the requirements for the traits class of the *polygon_area_2* function.
3. *OutputIterator::value_type* is equivalent to *std::pair<Dt::Point_2, Dt::Geom_traits::FT>*, i.e. a pair associating a point and its natural neighbor coordinate.

See Also

CGAL::linear_interpolation . . . . page 3993
CGAL::sibson_c1_interpolation . . . . page 3994
CGAL::surface_neighbor_coordinates_3 . . . . page 4016
CGAL::regular_neighbor_coordinates_2 . . . . page 4006

Implementation

This function computes the area of the sub-cells stolen from the Voronoi cells of the points in *dt* when inserting *p*. The total area of the Voronoi cell of *p* is also computed and returned by the function. If *p* lies outside the convex hull, the coordinate values cannot be computed and the third value of the result triple is set to *false*.

Function

CGAL::regular_neighbor_coordinates_2

Definition

The function *regular_neighbor_coordinates_2* computes natural neighbor coordinates, also called Sibson's coordinates, for weighted $2D$ points provided a two-dimensional regular triangulation and a (weighted) query point inside the convex hull of the vertices of the triangulation. We call these coordinates regular neighbor coordinates.

#include <CGAL/regular_neighbor_coordinates_2.h>

template < class Rt, class OutputIterator >
CGAL::Triple< OutputIterator, typename Rt::Geom_traits::FT, bool >

regular_neighbor_coordinates_2(Rt rt,
typename Rt::Weighted_point p,
OutputIterator out,
typename Rt::Face_handle start = typename Rt::Face_handle())

computes the regular neighbor coordinates for *p* with respect to the weighted points in the two-dimensional regular triangulation *rt*. The template class *Rt* should be of type *Regular_triangulation_2<Traits, Tds>*. The value type of the *OutputIterator* is a pair of *Rt::Weighted_point* and the coordinate value of type *Rt::Geom_traits::FT*. The sequence of point/-coordinate pairs that is computed by the function is placed starting at *out*. The function returns a triple with an iterator that is placed past-the-end of the resulting sequence of point/-coordinate pairs, the normalization factor of the coordinates and a Boolean value which is set to true iff the coordinate computation was successful, i.e. if *p* lies inside the convex hull of the points in *rt*.

template <class Rt, class OutputIterator, class EdgeIterator, class VertexIterator >
CGAL::Triple< OutputIterator, typename Traits::FT, bool >

regular_neighbor_coordinates_2(Rt rt,
typename Traits::Weighted_point p,
OutputIterator out,
EdgeIterator hole_begin,
EdgeIterator hole_end,
VertexIterator hidden_vertices_begin,

VertexIterator hidden_vertices_end)

The same as above. *hole_begin* and *hole_end* determines the iterator range over the boundary edges of the conflict zone of *p* in the triangulation *rt*. *hidden_vertices_begin* and *hidden_vertices_end* determines the iterator range over the hidden vertices of the conflict zone of *p* in*rt*. It is the result of the function *T.get_boundary_of_conflicts(p,std::back_inserter(hole), std::back_inserter(hidden_vertices), start)*, see *Regular_triangulation_2<Traits, Tds>*.

template <class Rt, class OutputIterator>
CGAL::Triple< OutputIterator, typename Rt::Geom_traits::FT, bool >

regular_neighbor_coordinates_2(Rt rt,
typename Rt::Vertex_handle vh,
OutputIterator out)

This function computes the regular neighbor coordinates of the point *vh->point()* with respect to the vertices of *rt* excluding *vh->point()*. The same as above for the remaining parameters.

Requirements

1. *Rt* are equivalent to the class *Regular_triangulation_2<Traits, Tds>*.
2. The traits class *Traits* of *Rt* is a model of the concept *RegularTriangulationTraits_2*. It provides the number type *FT* which is a model for *FieldNumberType* and it must meet the requirements for the traits class of the *polygon_area_2* function. A model of this traits class is *Regular_triangulation_euclidean_traits_2<K, Weight>*.
3. *OutputIterator::value_type* is equivalent to *std::pair<Rt::Weighted_point, Rt::Geom_traits::FT>*, i.e. a pair associating a point and its regular neighbor coordinate.

Implementation

This function computes the areas stolen from the Voronoi cells of points in *rt* by the insertion of *p*. The total area of the Voronoi cell of *p* is also computed and returned by the function. If *p* lies outside the convex hull, the coordinate values cannot be computed and the third value of the result triple is set to *false*.

See Also

CGAL::natural_neighbor_coordinates_2 . . . . page 4004

Function

CGAL::sibson_gradient_fitting

Definition

The function *sibson_gradient_fitting* approximates the gradient of a function at a point p given natural neighbor coordinates for p and its neighbors' function values. The approximation method is described in [Sib81]. Further functions are provided to fit the gradient for all data points that lie inside the convex hull of the data points. One function exists for each type of natural neighbor coordinates.

#include <CGAL/sibson_gradient_fitting.h>

template < class ForwardIterator, class Functor, class Traits>
typename Traits::Vector_d

sibson_gradient_fitting(ForwardIterator first,
ForwardIterator beyond,
typename std::iterator_traits<ForwardIterator>::value_type::second_type norm,
typename std::iterator_traits<ForwardIterator>::value_type::first_type p,
Functor f,
Traits traits)

This function estimates the gradient of a function at the point p given natural neighbor coordinates of p in the range $[$ *first*, *beyond* $)$ and the function values of the neighbors provided by the functor f. *norm* is the normalization factor of the barycentric coordinates.

Requirements

1. *ForwardIterator::value_type* is a pair of point/coordinate value, thus *ForwardIterator::value_type::first_type* is equivalent to a point and *ForwardIterator::value_type::second_type* is a number type.

2. *Functor::argument_type* must be equivalent to *ForwardIterator::value_type::first_type* and *Functor::result_type* is the function value type. It must provide a multiplication and addition operation with the type *ForwardIterator::value_type::second_type*.

3. *Traits* is a model of the concept *GradientFittingTraits*.

template < class Dt, class OutputIterator, class Functor, class Traits>

OutputIterator *sibson_gradient_fitting_nn_2(Dt dt, OutputIterator out, Functor f, Traits traits)*

estimates the function gradients at all vertices of *dt* that lie inside the convex hull using the coordinates computed by the function *natural_neighbor_coordinates_2*. *OutputIterator::value_type* is a pair associating a point to a vector. The sequence of point/gradient pairs computed by this function is placed starting at *out*. The function returns an iterator that is placed past-the-end of the resulting sequence. The requirements are the same as above. The template class *Dt* must be equivalent to *Delaunay_triangulation_2<Gt, Tds>* .

template < class Rt, class OutputIterator, class Functor, class Traits>
OutputIterator *sibson_gradient_fitting_rn_2(Rt rt, OutputIterator out, Functor f, Traits traits)*

estimates the function gradients at all vertices of *rt* that lie inside the convex hull using the coordinates computed by the function *regular_neighbor_coordinates_2*. *OutputIterator::value_type* is a pair associating a point to a vector. The sequence of point/gradient pairs computed by this function is placed starting at *out*. The function returns an iterator that is placed past-the-end of the resulting sequence. The requirements are the same as above. The template class *Rt* must be equivalent to *Regular_triangulation_2<Gt, Tds>*.

See Also

CGAL::linear_interpolation . . . . . . page 3993
CGAL::sibson_c1_interpolation . . . . . . page 3994
CGAL::farin_c1_interpolation . . . . . . page 3996
CGAL::quadratic_interpolation . . . . . . page 3998
CGAL::Interpolation_gradient_fitting_traits_2<K> . . . . . . page 4012
CGAL::natural_neighbor_coordinates_2 . . . . . . page 4004
CGAL::regular_neighbor_coordinates_2 . . . . . . page 4006
CGAL::surface_neighbor_coordinates_3 . . . . . . page 4016

Implementation

This function implements Sibson's gradient estimation method based on natural neighbor coordinates [Sib81].

Concept

GradientFittingTraits

Definition

The function *sibson_gradient_fitting* is parameterized by a traits class that defines the primitives used by the algorithm. The concept GradientFittingTraits defines this common set of requirements.

Types

GradientFittingTraits:: FT
The number type must follow the model *FieldNumberType*.

GradientFittingTraits:: Point_d
The point type on which the function is defined and interpolated.

GradientFittingTraits:: Vector_d
The corresponding vector type.

GradientFittingTraits:: Aff_transformation_d
defines a matrix type. Must provide the following member functions :
Aff_transformation tr.inverse () which gives the inverse transformation, and
Aff_transformation tr.transform(Vector v) which returns the multiplication of *tr* with *v*.

GradientFittingTraits:: Construct_vector_d
A constructor object for *Vector_d*. Provides :
Vector_d operator() (Point_d a, Point_d b) which produces the vector *b - a* and
Vector_d operator() (Null_vector NULL_VECTOR) which introduces the null vector.

GradientFittingTraits:: Construct_scaled_vector_d
Constructor object for *Vector_d*. Provides :
Vector_d operator() (Vector_d v,FT scale) which produces the vector *v* scaled by a factor *scale*.

GradientFittingTraits:: Construct_null_matrix_d
Constructor object for *Aff_transformation_d*. Provides :
Aff_transformation_d operator()() which introduces an affine transformation whose matrix has only zero entries.

GradientFittingTraits:: Construct_scaling_matrix_d
Constructor object for *Aff_transformation_d*. Provides :
Aff_transformation_d operator()(FT scale) which introduces a scaling by a scale factor *scale*.

GradientFittingTraits:: Construct_sum_matrix_d

Constructor object for *Aff_transformation_d*. Provides : *Aff_transformation_d operator()(Aff_transformation_d tr1, Aff_transformation_d tr2)* which returns the sum of the two matrices representing *tr1* and *tr2*.

GradientFittingTraits:: Construct_outer_product_d

Constructor object for *Aff_transformation_d*. Provides : *Aff_transformation_d operator()(Vector v)* which returns the outer product, i.e. the quadratic matrix $v^t v$.

Creation

GradientFittingTraits traits; default constructor.

Operations

The following functions that create instances of the above constructor object types must exist.

Construct_vector_d *traits.construct_vector_d_object()*

Construct_scaled_vector_d *traits.construct_scaled_vector_d_object()*

Construct_null_matrix_d *traits.construct_null_matrix_d_object()*

Construct_sum_matrix_d *traits.construct_sum_matrix_d_object()*

Construct_outer_product_d *traits.construct_outer_product_d_object()*

Has Models

CGAL::Interpolation_gradient_fitting_traits_2<K>

See Also

InterpolationTraits .... page 4001
CGAL::Interpolation_traits_2<K> .... page 4003
CGAL::sibson_gradient_fitting .... page 4008
CGAL::sibson_c1_interpolation .... page 3994
CGAL::farin_c1_interpolation .... page 3996
CGAL::quadratic_interpolation .... page 3998

Class

CGAL::Interpolation_gradient_fitting_traits_2<K>

Definition

Interpolation_gradient_fitting_traits_2<K> is a model for the concepts *InterpolationTraits* and *GradientFittingTraits*. It can be used to instantiate the geometric traits class of interpolation functions and of Sibson's gradient fitting function when applied on a function defined over a two-dimensional domain. The traits class is templated by a kernel class K.

#include <CGAL/Interpolation_gradient_fitting_traits_2.h>

Is Model for the Concepts

GradientFittingTraits InterpolationTraits

Types

typedef K::FT	*FT;*
typedef K::Point_2	*Point_d;*
typedef K::Vector_2	*Vector_d;*
typedef K::Aff_transformation_2	*Aff_transformation_d;*
typedef K::Construct_vector_2	*Construct_vector_d;*
typedef K::Construct_scaled_vector_2	*Construct_scaled_vector_d;*
typedef K::Compute_squared_distance_2	*Compute_squared_distance_d;*
typedef Construct_null_matrix_2<Aff_transformation_d>	*Construct_null_matrix_d;*
typedef Construct_scaling_matrix_2<Aff_transformation_d>	*Construct_scaling_matrix_d;*
typedef Construct_sum_matrix_2<Aff_transformation_d>	*Construct_sum_matrix_d;*
typedef Construct_outer_product_2<K>	*Construct_outer_product_d;*

Operations

Construct_scaled_vector_d	*traits.construct_scaled_vector_d_object() const*
Construct_vector_d	*traits.construct_vector_d_object() const*
Compute_squared_distance_d	*traits.compute_squared_distance_d_object() const*
Construct_null_matrix_d	*traits.construct_null_matrix_d_object() const*
Construct_scaling_matrix_d	*traits.construct_scaling_matrix_d_object() const*

Construct_sum_matrix_d	*traits.construct_sum_matrix_d_object() const*
Construct_outer_product_d	*traits.construct_outer_product_d_object() const*

See Also

InterpolationTraits . . . . page 4001
GradientFittingTraits . . . . page 4010
CGAL::Interpolation_traits_2<K> . . . . page 4003

CGAL::Voronoi_intersection_2_traits_3<K>

Definition

Voronoi_intersection_2_traits_3<K> is a model for the concept *RegularTriangulationTraits_2* and *InterpolationTraits*. It can be used to instantiate the geometric traits class of a two-dimensional regular triangulation. A three-dimensional plane is defined by a point and a vector that are members of the traits class. The triangulation is defined on 3*D* points. It is the regular triangulation of the input points projected onto the plane and each weighted with the negative squared distance of the input point to the plane. It can be shown that it is dual to the power diagram obtained by intersecting the three-dimensional Voronoi diagram of the input points with the plane. All predicates and constructions used in the computation of the regular triangulation are formulated on the three dimensional points without explicitly constructing the projected points and the weights. This reduces the arithmetic demands. The traits class is templated by a kernel class *K*.

#include <CGAL/Voronoi_intersection_2_traits_3.h>

Is Model for the Concepts

RegularTriangulationTraits_2 .... page 2524

Types

typedef K::RT	*Weight;*	
typedef K::FT	*FT;*	
typedef K::Point_3	*Point_2;*	
typedef K::Segment_3	*Segment_2;*	
typedef K::Triangle_3	*Triangle_2;*	
typedef K::Line_3	*Line_2;*	
typedef K::Ray_3	*Ray_2;*	
typedef K::Vector_3	*Vector_2;*	
typedef K::Construct_triangle_3	*Construct_triangle_2;*	
typedef K::Construct_ray_3	*Construct_ray_2;*	
typedef K::Compare_distance_3	*Compare_distance_2;*	
Compute_area_3<Rep>	*Compute_area_2;*	An instance of this function object class computes the square root of the result of *K::Compute_squared_area_3*. If the number type *FT* does not support the square root operation, the result is cast to *double* before computing the square root.

typedef Orientation_with_normal_plane_2_3<Rep>

Orientation_2;

typedef Side_of_plane_centered_sphere_2_3<Point_2>

Power_test_2;

typedef Construct_plane_centered_circumcenter_3<Point_2>

Construct_weighted_circumcenter_2;

typedef Compare_first_projection_3<Point_2>

Compare_x_2;

typedef Compare_second_projection_3<Point_2>

Compare_y_2;

Creation

Voronoi_intersection_2_traits_3<K> traits(typename K::Point_3 point=typename K::Point_3(), typename K::Vector_3 normal=NULL_VECTOR)

The plane associated to the traits class contains *point* and has as normal vector *normal*.

See Also

RegularTriangulationTraits_2 . . . . page 2524
CGAL::Regular_triangulation_2<Gt, Tds> . . . . page **??**
CGAL::regular_neighbor_coordinates_2 . . . . page 4006
CGAL::surface_neighbor_coordinates_3 . . . . page 4016

Function

CGAL::surface_neighbor_coordinates_3

Definition

The function *surface_neighbor_coordinates_3* computes natural neighbor coordinates for surface points associated to a finite set of sample points issued from the surface. The coordinates are computed from the intersection of the Voronoi cell of the query point p with the tangent plane to the surface at p. If the sampling is sufficiently dense, the coordinate system meets the properties described in the manual pages and in [BF02],[Flö03b]. The query point p needs to lie inside the convex hull of the projection of the sample points onto the tangent plane at p.

#include <CGAL/surface_neighbor_coordinates_3.h>

template <class OutputIterator, class InputIterator, class Kernel>
CGAL::Triple< OutputIterator, typename Kernel::FT, bool >

surface_neighbor_coordinates_3(InputIterator first,
InputIterator beyond,
typename Kernel::Point_3 p,
typename Kernel::Vector_3 normal,
OutputIterator out,
Kernel K)

The sample points $\mathcal{P}$ are provided in the range $[$*first*, *beyond*$)$. *InputIterator::value_type* is the point type *Kernel::Point_3*. The tangent plane is defined by the point p and the vector *normal*. The parameter K determines the kernel type that will instantiate the template parameter of *Voronoi_intersection_2_traits_3<K>*.
The natural neighbor coordinates for p are computed in the power diagram that results from the intersection of the $3D$ Voronoi diagram of $\mathcal{P}$ with the tangent plane. The sequence of point/coordinate pairs that is computed by the function is placed starting at *out*. The function returns a triple with an iterator that is placed past-the-end of the resulting sequence of point/coordinate pairs, the normalization factor of the coordinates and a Boolean value which is set to true iff the coordinate computation was successful, i.e. if p lies inside the convex hull of the projection of the points $\mathcal{P}$ onto the tangent plane.

template <class OutputIterator, class InputIterator, class ITraits>
CGAL::Triple< OutputIterator, typename ITraits::FT, bool >

surface_neighbor_coordinates_3(InputIterator first,
InputIterator beyond,
typename ITraits::Point_2 p,
OutputIterator out,

ITraits traits)

the same as above only that the traits class must be instantiated by the user. *ITraits* must be equivalent to *Voronoi_intersection_2_traits_3<K>*.

The next functions return, in addition, a second Boolean value (the fourth value of the quadruple) that certifies whether or not, the Voronoi cell of p can be affected by points that lie outside the input range, i.e. outside the ball centered on p passing through the furthest sample point from p in the range $[first, beyond)$. If the sample points are collected by a k-nearest neighbor or a range search query, this permits to check whether the neighborhood which has been considered is large enough.

template <class OutputIterator, class InputIterator, class Kernel>
CGAL::Quadruple< OutputIterator, typename Kernel::FT, bool, bool >

surface_neighbor_coordinates_certified_3(InputIterator first,
InputIterator beyond,
typename Kernel::Point_3 p,
typename Kernel::Vector_3 normal,
OutputIterator out,
Kernel K)

Similar to the first function. The additional fourth return value is *true* if the furthest point in the range $[first, beyond)$ is further away from p than twice the distance from p to the furthest vertex of the intersection of the Voronoi cell of p with the tangent plane defined by *(p,normal)*. It is *false* otherwise.

template <class OutputIterator, class InputIterator, class Kernel>
CGAL::Quadruple< OutputIterator, typename Kernel::FT, bool, bool >

surface_neighbor_coordinates_certified_3(InputIterator first,
InputIterator beyond,
typename Kernel::Point_3 p,
typename Kernel::FT max_distance,
OutputIterator out,
Kernel kernel)

The same as above except that this function takes the maximal distance from p to the points in the range $[first, beyond)$ as additional parameter.

template <class OutputIterator, class InputIterator, class ITraits>
CGAL::Quadruple< OutputIterator, typename ITraits::FT, bool, bool >

surface_neighbor_coordinates_certified_3(InputIterator first,
InputIterator beyond,
typename ITraits::Point_2 p,
OutputIterator out,
ITraits traits)

The same as above only that the traits class must be instantiated by the user and without the parameter *max_distance*. *ITraits* must be equivalent to *Voronoi_intersection_2_traits_3<K>*.

template <class OutputIterator, class InputIterator, class ITraits>
CGAL::Quadruple< OutputIterator, typename ITraits::FT, bool, bool >

surface_neighbor_coordinates_certified_3(InputIterator first, InputIterator beyond, typename ITraits::Point_2 p, typename ITraits::FT max_distance, OutputIterator out, ITraits traits)

The same as above with the parameter *max_distance*.

The next function allows to filter some potential neighbors of the query point p from $\mathcal{P}$ via its three-dimensional Delaunay triangulation. All surface neighbors of p are necessarily neighbors in the Delaunay triangulation of $\mathcal{P} \cup \{p\}$.

template < class Dt, class OutputIterator >
CGAL::Triple< OutputIterator, typename Dt::Geom_traits::FT, bool >

surface_neighbor_coordinates_3(Dt dt, typename Dt::Geom_traits::Point_3 p, typename Dt::Geom_traits::Vector_3 normal, OutputIterator out, typename Dt::Cell_handle start = typename Dt::Cell_handle())

computes the surface neighbor coordinates with respect to the points that are vertices of the Delaunay triangulation *dt*. The type *Dt* must be equivalent to *Delaunay_triangulation_3<Gt, Tds>*. The optional parameter *start* is used as a starting place for the search of the conflict zone. It may be the result of the call *dt.locate(p)*. This function instantiates the template parameter *ITraits* to be *Voronoi_intersection_2_traits_3<Dt::Geom_traits>*.

template < class Dt, class OutputIterator, class ITraits>
CGAL::Triple< OutputIterator, typename Dt::Geom_traits::FT, bool >

surface_neighbor_coordinates_3(Dt dt, typename Dt::Geom_traits::Point_3 p, OutputIterator out, ITraits traits, typename Dt::Cell_handle start = typename Dt::Cell_handle())

The same as above only that the parameter *traits* instantiates the geometric traits class. Its type *ITraits* must be equivalent to *Voronoi_intersection_2_traits_3<K>*.

Requirements

1. *Dt* is equivalent to the class *Delaunay_triangulation_3*.

2. *OutputIterator::value_type* is equivalent to *std::pair<Dt::Point_3, Dt::Geom_traits::FT>*, i.e. a pair associating a point and its natural neighbor coordinate.

3. *ITraits* is equivalent to the class *Voronoi_intersection_2_traits_3<K>*.

See Also

CGAL::linear_interpolation . . . . page 3993
CGAL::sibson_c1_interpolation . . . . page 3994
CGAL::farin_c1_interpolation . . . . page 3996
CGAL::Voronoi_intersection_2_traits_3<K> . . . . page 4014
CGAL::surface_neighbors_3 . . . . page 4020

Implementation

This functions construct the regular triangulation of the input points instantiated with *Voronoi_intersection_2_traits_3<Kernel>* or *ITraits* if provided. They return the result of the function call *regular_neighbor_coordinates_2* with the regular triangulation and *p* as arguments.

CGAL::surface_neighbors_3

Definition

Given a set of sample points issued from a surface and a query point p, the function *surface_neighbors_3* computes the neighbors of p on the surface within the sample points. If the sampling is sufficiently dense, the neighbors are provably close to the point p on the surface (cf. the manual pages and [BF02],[Flö03b]). They are defined to be the neighbors of p in the regular triangulation dual to the power diagram which is equivalent to the intersection of the Voronoi cell of the query point p with the tangent plane to the surface at p.

#include <CGAL/surface_neighbors_3.h>

template <class OutputIterator, class InputIterator, class Kernel>
OutputIterator surface_neighbors_3(InputIterator first,
InputIterator beyond,
typename Kernel::Point_3 p,
typename Kernel::Vector_3 normal,
OutputIterator out,
Kernel K)

The sample points $\mathcal{P}$ are provided in the range $[$*first*, *beyond*$)$. *InputIterator::value_type* is the point type *Kernel::Point_3*. The tangent plane is defined by the point p and the vector *normal*. The parameter K determines the kernel type that will instantiate the template parameter of *Voronoi_intersection_2_traits_3<K>*.
The surface neighbors of p are computed which are the neighbors of p in the regular triangulation that is dual to the intersection of the $3D$ Voronoi diagram of $\mathcal{P}$ with the tangent plane. The point sequence that is computed by the function is placed starting at *out*. The function returns an iterator that is placed past-the-end of the resulting point sequence.

template <class OutputIterator, class InputIterator, class ITraits>
OutputIterator surface_neighbors_3(InputIterator first,
InputIterator beyond,
typename ITraits::Point_2 p,
OutputIterator out,
ITraits traits)

the same as above only that the traits class must be instantiated by the user. *ITraits* must be equivalent to *Voronoi_intersection_2_traits_3<K>*.

The next functions return, in addition, a Boolean value that certifies whether or not, the Voronoi cell of p can be affected by points that lie outside the input range, i.e. outside the ball centered on p passing through the furthest sample point from p in the range $[$*first*, *beyond*$)$. If the sample points are collected by a k-nearest neighbor or a range search query, this permits to verify that a large enough neighborhood has been considered.

template <class OutputIterator, class InputIterator, class Kernel>

std::pair< OutputIterator, bool >

surface_neighbors_certified_3(InputIterator first,
InputIterator beyond,
typename Kernel::Point_3 p,
typename Kernel::Vector_3 normal,
OutputIterator out,
Kernel K)

Similar to the first function. The additional third return value is *true* if the furthest point in the range $[first, beyond)$ is further away from p than twice the distance from p to the furthest vertex of the intersection of the Voronoi cell of p with the tangent plane defined be *(p,normal)*. It is *false* otherwise.

template <class OutputIterator, class InputIterator, class Kernel>
std::pair< OutputIterator, bool >

surface_neighbors_certified_3(InputIterator first,
InputIterator beyond,
typename Kernel::Point_3 p,
typename Kernel::Vector_3 normal,
typename Kernel::FT max_distance,
OutputIterator out,
Kernel kernel)

The same as above except that this function takes the maximal distance from p to the points in the range $[first, beyond)$ as additional parameter.

template <class OutputIterator, class InputIterator, class ITraits>
std::pair< OutputIterator, bool >

surface_neighbors_certified_3(InputIterator first,
InputIterator beyond,
typename ITraits::Point_2 p,
OutputIterator out,
ITraits traits)

The same as above only that the traits class must be instantiated by the user. *ITraits* must be equivalent to *Voronoi_intersection_2_traits_3<K>*. There is no parameter *max_distance*.

template <class OutputIterator, class InputIterator, class ITraits>
std::pair< OutputIterator, bool >

surface_neighbors_certified_3(InputIterator first,
InputIterator beyond,
typename ITraits::Point_2 p,
typename ITraits::FT max_distance,
OutputIterator out,

ITraits traits)

The same as above with the parameter *max_distance*.

The next function allows to filter some potential neighbors of the query point p from $\mathcal{P}$ via its three-dimensional Delaunay triangulation. All surface neighbors of p are necessarily neighbors in the Delaunay triangulation of $\mathcal{P} \cup \{p\}$.

template < class Dt, class OutputIterator >
OutputIterator surface_neighbors_3(Dt dt,
typename Dt::Geom_traits::Point_3 p,
typename Dt::Geom_traits::Vector_3 normal,
OutputIterator out,
typename Dt::Cell_handle start = typename Dt::Cell_handle())

computes the surface neighbor coordinates with respect to the points that are vertices of the Delaunay triangulation *dt*. The type *Dt* must be equivalent to *Delaunay_triangulation_3<Gt, Tds>*. The optional parameter *start* is used for the used as a starting place for the search of the conflict zone. It may be the result of the call *dt.locate(p)*. This function instantiates the template parameter *ITraits* to be *Voronoi_intersection_2_traits_3<Dt::Geom_traits>*.

template < class Dt, class OutputIterator, class ITraits>
OutputIterator surface_neighbors_3(Dt dt,
typename ITraits::Point_2 p,
OutputIterator out,
ITraits traits,
typename Dt::Cell_handle start = typename Dt::Cell_handle())

The same as above only that the parameter *traits* instantiates the geometric traits class. Its type *ITraits* must be equivalent to *Voronoi_intersection_2_traits_3<K>*.

Requirements

1. *Dt* is equivalent to the class *Delaunay_triangulation_3*.
2. *OutputIterator::value_type* is equivalent to *Dt::Point_3*, i.e. a point type.
3. *ITraits* is equivalent to the class *Voronoi_intersection_2_traits_3<K>*.

See Also

CGAL::Voronoi_intersection_2_traits_3<K> .... page 4014
CGAL::surface_neighbor_coordinates_3 .... page 4016

Implementation

These functions compute the regular triangulation of the sample points and the point *p* using a traits class equivalent to *Voronoi_intersection_2_traits_3*<*K*>. They determine the neighbors of *p* in this triangulation. The functions which certify the result need to compute, in addition, the Voronoi vertices of the cell of *p* in this diagram.

Part XVI

Kinetic Data Structures

Chapter 71

Kinetic Data Structures

Daniel Russel

Contents

Lets say you want to maintain a sorted list of items (each item is associate with a real number key). You can imagine placing each of the items on the point on the real line corresponding to its key. Now, let the key for each item change continuously (i.e. no jumps are allowed). As long as no two (consecutive) items cross, the sorted order is intact. When two items cross, they need to be exchanged in the list and then the sorted order is once again correct. This is a trivial example of a kinetic data structure. The key observation is that the combinatorial structure which is maintained changes at discrete times (events) even though the basic building blocks are changing continuously.

This chapter describes a number of such kinetic data structures implemented using the Kinetic framework described in Chapter 72. We first, in Section 71.2 introduce kinetic data structures and sweepline algorithms. This section can be skipped if the reader is already familiar with the area. The next sections, Section 71.2.1 and Section 71.3 introduce the terms and give an overview of the framework. They are recommended reading for all readers, even if you are just using provided kinetic data structures. We then present kinetic data structures for Delaunay triangulations in two and three dimensions in Section 71.4.

If you are already familiar with kinetic data structures and know what you want to do, you might want to first take a look at the next section Section 71.1 which covers quick hints.

71.1 Quick Hints

This section gives quick answers to some questions people might have. It presumes knowledge of kinetic data structures and this framework.

How do I store extra information allow with, for example, a kinetic Point_2?

See the example *Kinetic_framework/defining_a_simulation_traits.cpp* to see how to define a new *SimulationTraits* class where the *ActiveObjectsTable* contains extra data along with the point.

Where is the best place to look if I want to write my own kinetic data structure?

We provide two simple kinetic data structures, first most trivial is *Kinetic_framework/trivial_kds.cpp* and a slightly more complicated one is:

#include <CGAL/Kinetic/Sort.h>

How can I use kinetic data structures to update Delaunay triangulations?

We are working on that one, but you will have to wait.

71.2 An Overview of Kinetic Data Structures and Sweep Algorithms

Kinetic data structures were first introduced in by Basch et. al. in 1997 [BGH97]. The idea stems from the observation that most, if not all, computational geometry structures are built using *predicates* — functions on quantities defining the geometric input (e.g. point coordinates), which return a discrete set of values. Many predicates reduce to determining the sign of a polynomial on the defining parameters of the primitive objects. For example, to test whether a point lies above or below a plane we compute the dot product of the point with the normal of the plane and subtract the plane's offset along the normal. If the result is positive, the point is above the plane, zero on the plane, negative below. The validity of many combinatorial structures built on top of geometric primitives can be verified by checking a finite number of predicates of the geometric primitives. These predicates, which collectively certify the correctness of the structure, are called *certificates*. For a Delaunay triangulation in three dimensions, for example, the certificates are one *InCircle* test per facet of the triangulation, plus a point plane orientation test for each facet or edge of the convex hull.

The kinetic data structures approach is built on top of this view of computational geometry. Let the geometric primitives move by replacing each of their defining quantities with a function of time (generally a polynomial). As time advances, the primitives trace out paths in space called *trajectories*. The values of the polynomial functions of the defining quantities used to evaluate the predicates now also become functions of time. We call these functions *certificate functions*. Typically, a geometric structure is valid when all predicates have a specific non-zero sign. In the kinetic setting, as long as the certificate functions maintain the correct sign as time varies, the corresponding predicates do not change values, and the original data structure remains correct. However, if one of the certificate functions changes sign, the original structure must be updated, as well as the set of certificate functions that verify it. We call such occurrences *events*.

Maintaining a kinetic data structure is then a matter of determining which certificate function changes sign next, i.e. determining which certificate function has the first real root that is greater than the current time, and then updating the structure and the set of certificate functions. In addition, the trajectories of primitives are allowed to change at any time, although C^0-continuity of the trajectories must be maintained. When a trajectory update occurs for a geometric primitive, all certificates involving that primitive must be updated. We call the collection of kinetic data structures, primitives, event queue and other support structures a *simulation*.

Sweepline algorithms for computing arrangements in d dimensions easily map on to kinetic data structures by taking one of the coordinates of the ambient space as the time variable. The kinetic data structure then maintains the arrangement of a set of objects defined by the intersection of a hyperplane of dimension $d-1$ with the objects whose arrangement is being computed.

Time is one of the central concepts in a kinetic simulation. Just as static geometric data structures divide the continuous space of all possible inputs (as defined by sets of coordinates) into a discrete set of combinatorial structures, kinetic data structures divide the continuous time domain into a set of disjoint intervals. In each interval the combinatorial structure does not change, so, in terms of the combinatorial structure, all times in the interval are equivalent. We capitalize on this equivalence in the framework in order to simplify computations. If the primitives move on polynomial trajectories and the certificates are polynomials in the coordinates, then events occur at real roots of polynomials of time. Real numbers, which define the endpoints of the interval, are more expensive to compute with than rational numbers, so performing computations at a rational number inside the interval is preferable whenever possible. See Section 72.1.4 for an example of where this equivalence is exploited.

71.2.1 Terms Used

primitive The basic geometric types, e.g., the points of a triangulation. A primitive has a set of *coordinates*.

combinatorial structure A structure built on top of the primitives. The structure does not depend directly on the coordinates of the primitives, only on relationships between them.

trajectory The path traced out by a primitive as time passes. In other words how the coordinates of a primitive change with time.

snapshot The position of all the primitives at a particular moment in time.

static Having to do with geometric data structures on non-moving primitives.

predicate A function which takes the coordinates of several primitives from a snapshot as input and produces one of a discrete set of outputs.

certificate One of a set of predicates which, when all having the correct values, ensure that the combinatorial structure is correct.

certificate function A function of time which is positive when the corresponding certificate has the correct value. When the certificate function changes sign, the combinatorial structure needs to be updated.

event When a certificate function changes sign and the combinatorial structure needs to be updated.

71.3 An Overview of the Kinetic Framework

The provided kinetic data structures are implemented on top of the Kinetic framework presented in Chapter 72. It is not necessary to know the details of the framework, but some familiarity is useful. Here we presented a quick overview of the framework.

The framework is structured around five main concepts. See Figure 71.1 for a schematic of how a kinetic data structure interacts with the various parts. The main concepts are

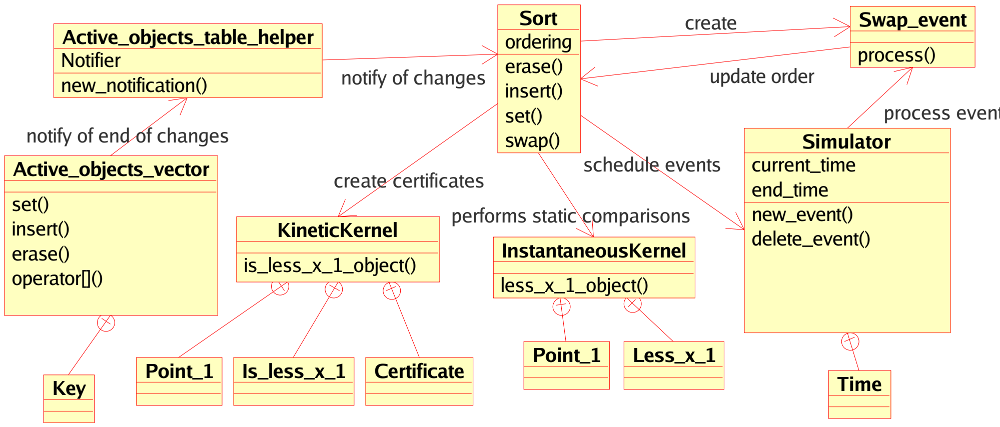


Figure 71.1: The figure shows the interaction between the *Kinetic::Sort<Traits, Visitor>* kinetic data structure and the various pieces of our package. Other, more complicated, kinetic data structures will also use the *Kinetic::InstantaneousKernel* in order to insert/remove geometric primitives and audit themselves. *Kinetic::Sort< Traits, Visitor>* uses the sorting functionality in the STL instead.

- the *Kinetic::Simulator*. Models of this concept process events in the correct order and audit kinetic data structures. There should be one instance of a model of this concept per simulation.
- the *Kinetic::Kernel*. The structure of a *Kinetic::Kernel* is analogous to the static CGAL (i.e., non-kinetic) kernels in that it defines a set of primitives and functors which generate certificates from the primitives.
- the *Kinetic::ActiveObjectsTable*. Models of this concept hold a collection of kinetic primitives in a centralized manner. This structure centralizes management of the primitives in order to properly disseminate notifications when trajectories change, new primitives are added or primitives are deleted. There is generally one instance of a model of this concept per simulation.
- the *Kinetic::InstantaneousKernel*. Models of this concept allow existing non-kinetic CGAL data structures to be used on a snapshot of kinetic data. As a result, pre-existing static structures can be used to initialize and audit kinetic data structures.
- the *Kinetic::FunctionKernel*. This concept is the computational kernel of our framework. Models of this concept are responsible for representing, generating and manipulating the motional and certificate functions and their roots. It is this concept that provides the kinetic data structures framework with the necessary algebraic operations for manipulating event times. The *Kinetic::FunctionKernel* is discussed in detail in Section 72.2.

For simplicity, we added an additional concept, that of *Kinetic::SimulationTraits*, which wraps together a particular set of choices for the above concepts and is responsible for creating instances of each of the models. As a user of existing kinetic data structures, this is the only framework object you will have to create. The addition of this concept reduces the choices the user has to make to picking the dimension of the ambient space and choosing between exact and inexact computations. The model of *Kinetic::SimulationTraits* creates an instance each of the *Kinetic::Simulator* and *Kinetic::ActiveObjectsTable*. Handles for these instances as well as instances of the *Kinetic::Kernel* and *Kinetic::InstantaneousKernel* can be requested from the simulation traits class. Both the *Kinetic::Kernel* and the *Kinetic::Simulator* use the *Kinetic::FunctionKernel*, the former to find certificate failure times and the later to operate on them. For technical reasons, each supplied model of *Kinetic::SimulationTraits* also picks out a particular type of kinetic primitive which will be used by the kinetic data structures.

71.4 Using Kinetic Data Structures

There are five provided kinetic data structures. They are

Kinetic::Sort<Traits, Visitor> maintain a list of points sorted by x-coordinate.

Kinetic::Delaunay_triangulation_2<Traits, Visitor, Triangulation> maintain the Delaunay triangulation of a set of two dimensional points

Kinetic::Delaunay_triangulation_3<Traits,Visitor, Triangulation> maintain the Delaunay triangulation of a set of three dimensional points.

Kinetic::Regular_triangulation_3<Traits, Visitor, Triangulation> maintain the regular triangulation of a set of waiting three dimensional points.

Kinetic::Enclosing_box_2<Traits>, *Kinetic::Enclosing_box_3<Traits>* restrict points to stay within a box by bouncing them off the walls.

71.4.1 A Simple Example

Using a kinetic data structure can be as simple as the following:

```
#include <CGAL/Kinetic/basic.h>

#include <CGAL/Kinetic/Exact_simulation_traits.h>
#include <CGAL/Kinetic/Insert_event.h>
#include <CGAL/Kinetic/Sort.h>

int main()
{
    typedef CGAL::Kinetic::Exact_simulation_traits Traits;

    typedef CGAL::Kinetic::Insert_event<Traits::Active_points_1_table>
   Insert_event;
    typedef Traits::Active_points_1_table::Data Moving_point;
    typedef CGAL::Kinetic::Sort<Traits> Sort;
    typedef Traits::Simulator::Time Time;

    Traits tr(0,100000);
    Sort sort(tr);
    Traits::Simulator::Handle sp= tr.simulator_handle();

    std::ifstream in("data/points_1");
    in  >> *tr.active_points_1_table_handle();

    while (sp->next_event_time() != sp->end_time()) {
        sp->set_current_event_number(sp->current_event_number()+1);
    }

    return EXIT_SUCCESS;
}
```

File: examples/Kinetic_data_structures/Kinetic_sort.cpp

Using the other kinetic data structures is substantially identical. Please see the appropriate files in the demo/Kinetic_data_structures directory.

In the example, first the *Kinetic::SimulationTraits* object is chosen (in this case one that supports exact computations). Then the kinetic data structure is defined using the chosen traits object and a visitor class which logs changes to the sorted list. Next, instances of the two are created and a set of points is read from a file. Then, the simulator is instructed to process all the events until the end of the simulation. Finally, a record of what happened is printed to the terminal.

Several important things happen behind the scenes in this example. First, the *Kinetic::ActiveObjectsTable* which holds the moving points notifies the kinetic data structure that new points have been added to the simulation. Second, the *Kinetic::Sort<Traits,Visitor>* kinetic data structure registers its events with the *Kinetic::Simulator* by providing a time and a proxy object for each event. When a particular event occurs, the *Kinetic::Simulator* calls a function on the proxy object which in turn updates the kinetic data structure.

The example illustrates how to monitor the supplied data structures as they evolve by using a *Kinetic::SortVisitor* object—a small class whose methods are called whenever the kinetic data structure changes. Hooks for such visitor concepts are provided for all of the shipped kinetic data structures. In the case of kinetic sorting, the visitor's methods are called every time a new point is inserted in the sorted list, when one is removed, or when two points are swapped in the sorted order.

The visitor concept is quite powerful, allowing us, for example, to implement a data structure for computing and storing two-dimensional arrangements of x-monotone curves on top of the *Kinetic::Sort<Traits, Visitor>* data structure using about 60 lines of code. This sweepline code is presented in Section 71.4.4.

71.4.2 Creating Kinetic Primitives

One key part of the framework not shown is how to create kinetic primitives (rather than just reading them in from a file). There are two ways to construction the necessary motion functions (which are models of *Kinetic::FunctionKernel::Function*). The first is to create an array of polynomial coeffients and simply call the constructor as in:

```
typedef Traits::Kinetic_kernel::Motion_function F;
std::vector<F::NT> coefs;
coefs.push_back(F::NT(1.0));
coefs.push_back(F::NT(2.0));
F x(coefs.begin(), coefs.end());
```

A slightly more flexible way is to use a *Kinetic::FunctionKernel::ConstructFunction* object. To do this do the following:

```
typedef Traits::Kinetic_kernel::Function_kernel::Construct_function
CF; typedef Traits::Kinetic_kernel::Motion_function F; CF cf; F
x=cf(F::NT(1.0), F::NT(2.0));
```

The *Kinetic::FunctionKernel::ConstructFunction* can be passed (almost) an number of arguments and will construct a polynomial with those arguments are coefficients.

Once the motion functions are constructed, constructing the primitive is just like constructing the corresponding static object.

```
typedef Traits::Kinetic_kernel::Point_1 Point_1;
Point_1 p(x);
```

71.4.3 Visualization of Kinetic Data Structures

The framework includes Qt widgets for displaying kinetic data structures in two and three dimensions. The following example shows using the two dimensional widget with a Delaunay triangulation:

```
#include <CGAL/Kinetic/Exact_simulation_traits.h>
#include <CGAL/Kinetic/Delaunay_triangulation_2.h>
#include <CGAL/Kinetic/Enclosing_box_2.h>
#include <CGAL/Kinetic/IO/Qt_moving_points_2.h>
#include <CGAL/Kinetic/IO/Qt_triangulation_2.h>
#include <CGAL/Kinetic/IO/Qt_widget_2.h>

int main(int argc, char *argv[]) {
    using namespace CGAL::Kinetic;
    typedef Exact_simulation_traits Traits;
    typedef Delaunay_triangulation_2<Traits> Del_2;
    typedef Enclosing_box_2<Traits> Box_2;
    typedef Qt_widget_2<Traits::Simulator> Qt_widget;
    typedef Qt_moving_points_2<Traits, Qt_gui> Qt_mps;
    typedef Qt_triangulation_2<Del_2, Qt_widget, Qt_mps> Qt_dt2;

    // create a simulation traits and add two KDSs:
    // a kinetic Delaunay triangulation and an enclosing box;
    // the moving points bounce against the walls of the enclosing box
    Traits tr;
    Box_2::Handle box = new Box_2(tr);
    Del_2::Handle kdel = new Del_2(tr);

    // register the simulator, set of moving points and
    // Delaunay triangulation with the kinetic Qt widget
    Qt_widget::Handle qt_w = new Qt_widget(argc, argv,
   tr.simulator_handle());
    Qt_mps::Handle qt_mps = new Qt_mps(qt_w, tr);
    Qt_dt2::Handle qt_dt2 = new Qt_dt2(kdel, qt_w, qt_mps);

    // read the trajectories of the moving points
    //  the simulation traits automatically inserts them in the two KDSs
    // and schedules the appropriate kinetic events; as in the kinetic
    // sorting example this is done with appropriate notifications
    std::ifstream in("data/points_2");
    in  >> *tr.active_points_2_table_handle();

    // run the interactive kinetic simulation
    return qt_w->begin_event_loop();
};
```

The example shows how to use a number of additional features of the framework. First, it shows that two kinetic data structures (*Kinetic::Delaunay_triangulation_2<Traits, Triangulation>* and *Kinetic::Enclosing_box_*

2<Traits>) can coexist on the same set of points without any extra effort. Both interact with the moving points through the active objects table, and never need to directly interact with one another. Second, objects (like `qt_w`, `qt_mps` and `qt_dt2`) are all stored by using reference counted handles (`Object::Handle`). This allows them to share references to one another without the user having to worry about memory management and order of deletion. For example, the *Kinetic::Qt_triangulation_2<KineticDelaunay_2, QtWidget_2, Qt_moving_points_2>* object needs a handle to the kinetic triangulation, in order to get the structure to display, and a handle to the *Active_points_1_table* to get the coordinates of the points.

Finally, the example shows how to use the graphical interface elements provided, see Figure 71.3. Our package includes `Qt` widgets for displaying kinetic geometry in two and three dimensions. In addition to being able to play and pause the simulation, the user can step through events one at a time and reverse the simulation to retrace what had happened. The three-dimensional visualization support is based on the Coin library http://www.coin3d.org.

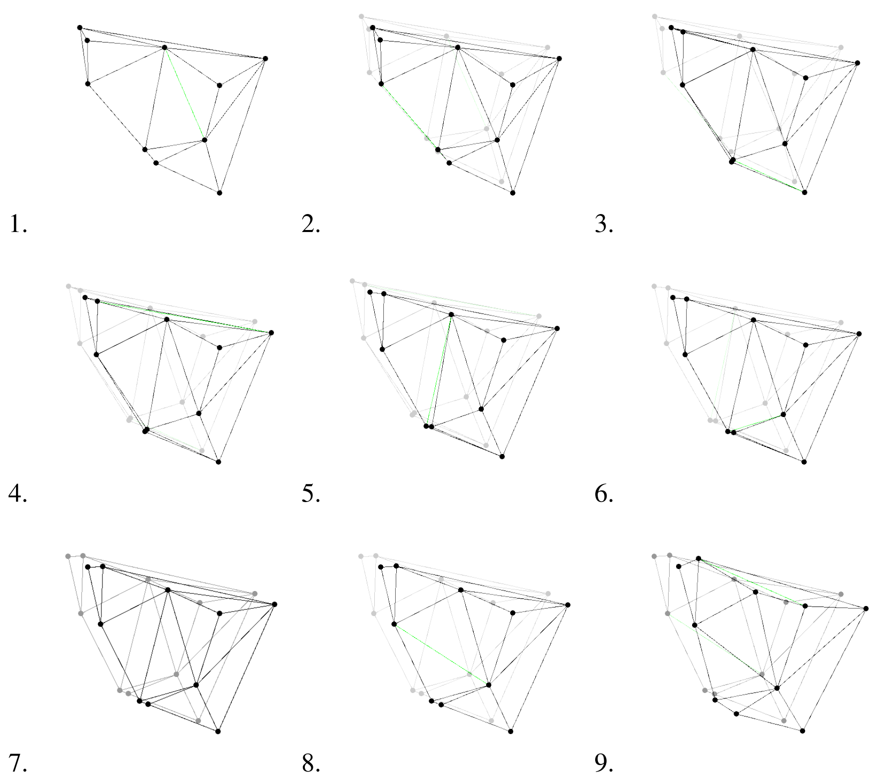


Figure 71.2: *Some events from a Delaunay triangulation kinetic data structure:* The state of the two dimensional Delaunay triangulation immediately following the first events is shown. Green edges are ones which were just created. The pictures are screen shots from *demo/Kinetic_data_structures/Kinetic_Delaunay_triangulation_2.cpp*.

71.4.4 Extending Kinetic Data Structures

Here we present a simple example that uses the *Kinetic::Sort<Traits, Visitor>* kinetic data structure to compute an arrangement of algebraic functions. It wraps the sorting data structure and uses a visitor to monitor changes and map them to corresponding features in the arrangement. To see an example using this kinetic data structure read the example at examples/Kinetic_data_structures/Kinetic_sweepline.cpp.

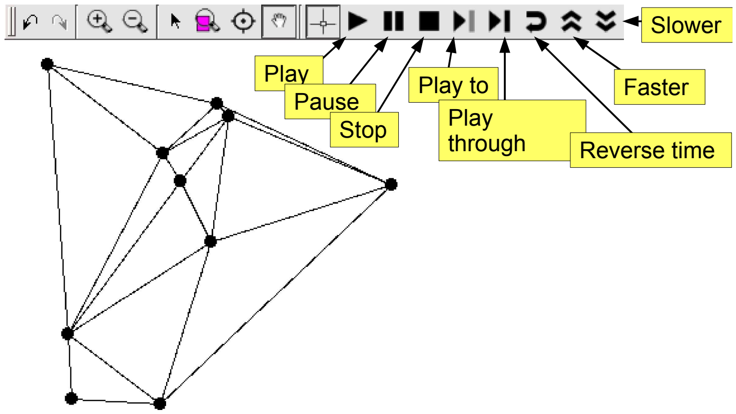


Figure 71.3: The figure shows the graphical user interface for controlling two-dimensional kinetic data structures. It is built on top of the *Qt_widget* and adds buttons to play, pause, step through and run the simulation backwards.

First we define the visitor class. An object of this type is passed to the *Kinetic::Sort<Traits, Visitor>* data structure and turns events into calls on the arrangement structure. This class has to be defined externally since the arrangement will inherit from the sorting structure.

```
template <class Arrangement>
struct Arrangement_visitor: public Kinetic::Sort_visitor_base
{
  Arrangement_visitor(Arrangement *a):p_(a){}
  template <class Vertex_handle>
  void remove_vertex(Vertex_handle a) {
    p_->erase(a);
  }
  template <class Vertex_handle>
  void create_vertex(Vertex_handle a) {
    p_->insert(a);
  }
  template <class Vertex_handle>
  void after_swap(Vertex_handle a, Vertex_handle b) {
    p_->swap(a, b);
  }
  Arrangement *p_;
};
```

Now we define the actual arrangement data structure.

```
template <class TraitsT>
class Planar_arrangement:
  public Kinetic::Sort<TraitsT,
                       Arrangement_visitor<Planar_arrangement<TraitsT> >
   > {
  typedef TraitsT Traits;
  typedef Planar_arrangement<TraitsT> This;
  typedef typename Kinetic::Sort<TraitsT,
                                 Arrangement_visitor<This> > Sort;
  typedef Arrangement_visitor<This> Visitor;
```

```
  typedef typename Traits::Active_objects_table::Key Key;

public:
  typedef CGAL::Exact_predicates_inexact_constructions_kernel::Point_2
    Approximate_point;
  typedef std::pair<int,int> Edge;
  typedef typename Sort::Vertex_handle Vertex_handle;

  // Register this KDS with the MovingObjectTable and the Simulator
  Planar_arrangement(Traits tr): Sort(tr, Visitor(this)) {}

  Approximate_point vertex(int i) const
  {
    return approx_coords_[i];
  }

  size_t vertices_size() const
  {
    return approx_coords_.size();
  }

  typedef std::vector<Edge >::const_iterator Edges_iterator;
  Edges_iterator edges_begin() const
  {
    return edges_.begin();
  }
  Edges_iterator edges_end() const
  {
    return edges_.end();
  }

  void insert(Vertex_handle k) {
    last_points_[*k]=new_point(*k);
  }

  void swap(Vertex_handle a, Vertex_handle b) {
    int swap_point= new_point(*a);
    edges_.push_back(Edge(swap_point, last_points_[*a]));
    edges_.push_back(Edge(swap_point, last_points_[*b]));
    last_points_[*a]= swap_point;
    last_points_[*b]= swap_point;
  }

  void erase(Vertex_handle a) {
    edges_.push_back(Edge(last_points_[*a], new_point(*a)));
  }

  int new_point(typename Traits::Active_objects_table::Key k) {
    double tv=
   CGAL::to_double(Sort::traits().simulator_handle()->current_time());
    double dv=
   CGAL::to_double(Sort::traits().active_objects_table_handle()->at(k).x()(tv));
    approx_coords_.push_back(Approximate_point(tv, dv));
    return approx_coords_.size()-1;
```

```
  }

  std::vector<Approximate_point > approx_coords_;
  std::map<Key, int> last_points_;
  std::vector<Edge> edges_;

};
```

Finally, we have to set everything up. To do this we use some special event classes: *Kinetic::Insert_event<ActiveObjectsTable>* and *Kinetic::Erase_event<ActiveObjectsTable>*. These are events which can be put in the event queue which either insert a primitive into the set of active objects or remove it. Using these, we can allow curves in the arrangement to begin or end in arbitrary places.

```
typedef CGAL::Kinetic::Insert_event<Traits::Active_points_1_table>
   Insert_event;
typedef CGAL::Kinetic::Erase_event<Traits::Active_points_1_table>
   Erase_event;
do {
  NT begin, end;
  Point function;
  // initialize the function and the beginning and end somewhere
  tr.simulator_handle()->new_event(Time(begin),
                                   Insert_event(function,
   tr.active_points_1_table_handle()));
  tr.simulator_handle()->new_event(Time(end),

   Erase_event(Traits::Active_points_1_table::Key(num),

   tr.active_points_1_table_handle()));
  ++num;
} while (true);
```

Kinetic Data Structures Reference Manual

Daniel Russel

71.5 Classified Reference Pages

Kinetic data structures are a way of adding motion to classical geometric data structures. CGAL provides several prepackaged kinetic data structures. Here we present those kinetic data structures and the helper classes that allow their activity to be monitored.

Sorting

Delaunay Triangulation in 2D

Delaunay and Regular Triangulations in 3D

Support Classes

71.6 Alphabetical List of Reference Pages

CGAL::Kinetic::Delaunay_triangulation_2<Traits, Visitor, Triangulation>

Definition

The class *Kinetic::Delaunay_triangulation_2<Traits, Visitor, Triangulation>* maintains a Delaunay triangulation on top of the points contained in a *Kinetic::ActiveObjectsTable*. It has one main method of interest, *triangulation()*, which returns the triangulation it is maintaining.

Note that the Delaunay triangulation is fully dynamic as it tracks points added to and removed from the *Kinetic::ActiveObjectsTable*.

The class *Kinetic::Qt_triangulation_2<KineticTriangulation_2, QtWidget_2, QtMovingPoints_2>* displays a kinetic Delaunay triangulation using the Qt widget.

This class is a good example of a simple, but non-trivial, kinetic data structure.

The *Triangulation* template parameter must be a model of *CGAL::Delaunay_triangulation_2<Traits, Tds>* which uses *Traits::Default_instantaneous_kernel* as its geometric traits and a *Tds* whose face inherits from *Kinetic::Delaunay_triangulation_face_base_2<Traits, Base>*.

The optional *Visitor* parameter takes a model of *Kinetic::DelaunayTriangulationVisitor_2*. Methods on this object will be called whenever the triangulation changes.

#include <CGAL/Kinetic/Delaunay_triangulation_2.h>

Is Model for the Concepts

Ref_counted<T>

Types

Kinetic::Delaunay_triangulation_2<Traits, Visitor, Triangulation>:: Triangulation

The template argument triangulation.

Kinetic::Delaunay_triangulation_2<Traits, Visitor, Triangulation>:: Visitor

The template argument for the visitor.

Creation

Kinetic::Delaunay_triangulation_2<Traits, Visitor, Triangulation> dt(Traits tr);

Maintain the Delaunay triangulation of the points in *tr.active_points_2_handle()*.

Operations

Triangulation *dt.triangulation() const*

Access the triangulation that is maintained.

Visitor& *dt.visitor()* Access the visitor.

Vertex_handle *dt.insert(Point_key k)*

Insert the point.

void *dt.erase(Vertex_handle h)*

Erase the vertex.

See Also

Kinetic::DelaunayTriangulationVisitor_2, *Kinetic::Delaunay_triangulation_default_visitor_2*, *Kinetic::Delaunay_triangulation_recent_edges_visitor_2<Triangulation>*, *Kinetic::Delaunay_triangulation_event_log_visitor_2*, *Kinetic::Qt_Delaunay_triangulation_2*.

Class

CGAL::Kinetic::Delaunay_triangulation_3<Traits, Visitor, Triangulation>

Definition

The class *Kinetic::Delaunay_triangulation_3<Traits, Visitor, Triangulation>* maintains a Delaunay triangulation on top of the points contained in a *Kinetic::ActiveObjectsTable*. It has one main method of interest. *triangulation()* which returns the triangulation it is maintaining. In addition, as an optimisation, you can turn on and off whether it is currently maintaining its certificates. This allows a large number of changes to the underlying points to be made at one time without recomputing the certificates each time a single point changes.

Note that the Delaunay triangulation is fully dynamic as it tracks points added to and removed from the *Kinetic::ActiveObjectsTable*.

The class *Kinetic::Qt_triangulation_3<Traits>*, included as part of the demo code, displays a kinetic Delaunay triangulation in three dimensions using the Coin library.

The optional *Visitor* template argument is a model of *Kinetic::DelaunayTriangulationVisitor_3* and can be used to monitor changes in the kinetic data structure.

The optional *Triangulation* template argument must be a model of a *CGAL::DelaunayTriangulation_3* which uses *Traits::Default_instantaneous_kernel* as its geometric traits and has *Kinetic::Delaunay_triangulation_cell_base_3<Traits, Base>* a the cell type.

#include <CGAL/Kinetic/Delaunay_triangulation_3.h>

Types

Kinetic::Delaunay_triangulation_3<Traits, Visitor, Triangulation>:: Triangulation

The template argument.

Kinetic::Delaunay_triangulation_3<Traits, Visitor, Triangulation>:: Visitor

The template argument.

Creation

Kinetic::Delaunay_triangulation_3<Traits, Visitor, Triangulation> dt(Traits tr);

Maintain the Delaunay triangulation of the points in *tr.active_points_3_handle()*.

Operations

*const Triangulation**	*dt.triangulation()*	Access the triangulation that is maintained.

bool	*dt.has_certificates()*	This method returns true if the *Kinetic::Delaunay_triangulation_3<Traits, Visitor, Triangulation>* is currently maintaining certificates for a Delaunay triangulation.
void	*dt.set_has_certificates(bool tf)*	This method allows you to control whether the triangulation is maintaining certificates.
Visitor&	*dt.visitor()*	Access the visitor.

See Also

Kinetic::Regular_triangulation_3<Traits, Triangulation, Visitor>, *Kinetic::Delaunay_triangulation_2<Traits, Triangulation, Visitor*, *Kinetic::Delaunay_triangulation_visitor_base_3*, *Kinetic::Delaunay_triangulation_event_log_visitor_3*.

Class

CGAL::Kinetic::Delaunay_triangulation_cell_base_3<Traits, Base>

Definition

This is the base class for faces used by *Kinetic::Delaunay_triangulation_3<Traits, Triangulation, Visitor> ::Triangulation*.

#include <CGAL/Kinetic/Delaunay_triangulation_cell_base_3.h>

CGAL::Kinetic::Delaunay_triangulation_event_log_visitor_2

Class

Definition

The concept *Kinetic::Delaunay_triangulation_event_log_visitor_2* provides a model of *Kinetic::DelaunayTriangulationVisitor_2* and *Kinetic::EventLogVisitor* which logs edge flip events.

Is Model for the Concepts

Kinetic::DelaunayTriangulationVisitor_2, *Kinetic::EventLogVisitor*

See Also

Kinetic::Delaunay_triangulation_2<Traits, Triangulation, Visitor>

CGAL::Kinetic::Delaunay_triangulation_event_log_visitor_3

Definition

The concept *Kinetic::Delaunay_triangulation_event_log_visitor_3* provides a model of *Kinetic::DelaunayTriangulationVisitor_3* and *Kinetic::EventLogVisitor* which logs edge and facet flip events.

Is Model for the Concepts

Kinetic::DelaunayTriangulationVisitor_3, *Kinetic::EventLogVisitor*

See Also

Kinetic::Delaunay_triangulation_3<Traits, Triangulation, Visitor>

CGAL::Kinetic::Delaunay_triangulation_face_base_2<Traits, Base>

Definition

This is the base class for faces used by the triangulation used in *Kinetic::Delaunay_triangulation_2<Traits, Triangulation, Visitor>*.

#include <CGAL/Kinetic/Delaunay_triangulation_face_base_2.h>

See Also

Kinetic:Delaunay_triangulation_2<Traits, Triangulation, Visitor>

Class

CGAL::Kinetic::Delaunay_triangulation_recent_edges_visitor_2< Triangulation>

Definition

The concept *Kinetic::Delaunay_triangulation_recent_edges_visitor_2<Triangulation>* provides a model of *Kinetic::DelaunayTriangulationVisitor_2* which tracks which edges were created in the most recent change.

Is Model for the Concepts

Kinetic::DelaunayTriangulationVisitor_2

Creation

Kinetic::Delaunay_triangulation_recent_edges_visitor_2<Triangulation> a;

default constructor.

Kinetic::Delaunay_triangulation_recent_edges_visitor_2<Triangulation>:: iterator

The iterator through the recently created edges.

Operations

iterator	*a.begin() const*	Begin iteration through the recent edges.
iterator	*a.end() const*	End iteration through the recent edges.
bool	*a.contains(Triangulation::Edge) const*	Returns true if this edge exists in the set.

See Also

Kinetic::Delaunay_triangulation_2<Traits, Triangulation, Visitor>

Kinetic::DelaunayTriangulationVisitor_2

Definition

This concept is for proxy objects which get notified when a kinetic Delaunay triangulation changes.

Operations

void	*v.pre_remove_vertex(Vertex_handle)*	
		The vertex is about to be deleted.
void	*v.post_remove_vertex(Point_key)*	
		The vertex was just removed.
void	*v.pre_insert_vertex(Point_key)*	
		The vertex is about to be inserted into the cell (although the *Cell_handle* might be *NULL*).
void	*v.post_insert_vertex(Vertex_handle)*	
		The vertex was just inserted.
void	*v.change_vertex(Vertex_handle)*	
		The trajectory of the point at the vertex changed.
void	*v.create_face(Face_handle f)*	
		The face has just been made.
void	*v.destroy_face(Face_handle f)*	
		The face is about to be destroyed.
void	*v.pre_flip(Edge)*	The edge is about to be flipped.
void	*v.post_flip(Edge)*	The edge was just created with a flip.

Has Models

Kinetic::Delaunay_triangulation_visitor_base_2, *Kinetic::Delaunay_triangulation_recent_edges_visitor_2< Triangulation>*, *Kinetic::Delaunay_triangulation_event_log_visitor_2*

Kinetic::DelaunayTriangulationVisitor_3

Definition

This concept is for proxy objects which get notified when a kinetic Delaunay triangulation changes.

Operations

void *v.pre_remove_vertex(Vertex_handle)*

The vertex is about to be deleted.

void *v.post_remove_vertex(Point_key, Cell_handle)*

The vertex was just removed.

void *v.pre_insert_vertex(Point_key, Cell_handle)*

The vertex is about to be inserted into the cell (although the *Cell_handle* might be *NULL*).

void *v.post_insert_vertex(Vertex_handle)*

The vertex was just inserted.

void *v.change_vertex(Vertex_handle)*

The trajectory of the point at the vertex changed.

void *v.create_cell(Cell_handle)*

The cell was just created and initialized.

void *v.destroy_cell(Cell_handle)*

The cell is about to be removed.

void *v.pre_edge_flip(Edge)*

The edge is about to be flipped.

void *v.post_edge_flip(Facet)*

The facet was just created with a flip.

void *v.pre_facet_flip(Facet)*

The facet is about to be flipped.

void *v.post_facet_flip(Edge)*

The edge was just created with a flip.

Has Models

Kinetic::Delaunay_triangulation_visitor_base_3, *Kinetic::Delaunay_triangulation_recent_edges_visitor_3< Triangulation>*, *Kinetic::Delaunay_triangulation_event_log_visitor_3*

Class

CGAL::Kinetic::Delaunay_triangulation_visitor_base_2

Definition

The concept *Kinetic::Delaunay_triangulation_visitor_base_2* provides a model of *Kinetic::DelaunayTriangulationVisitor_2*. You can extend this class if you only want to implement a few methods from *Kinetic::DelaunayTriangulationVisitor_2*.

Is Model for the Concepts

Kinetic::DelaunayTriangulationVisitor_2

Creation

Kinetic::Delaunay_triangulation_visitor_base_2 a;

default constructor.

See Also

Kinetic::Delaunay_triangulation_2<Traits, Triangulation, Visitor>

CGAL::Kinetic::Delaunay_triangulation_visitor_base_3

Class

Definition

The concept *Kinetic::Delaunay_triangulation_visitor_base_3* provides a model of *Kinetic::DelaunayTriangulationVisitor_3*. You can extend this class if you only want to implement a few methods from *Kinetic::DelaunayTriangulationVisitor_3*.

Is Model for the Concepts

Kinetic::DelaunayTriangulationVisitor_3

Creation

Kinetic::Delaunay_triangulation_visitor_base_3 a;

default constructor.

See Also

Kinetic::Delaunay_triangulation_3<Traits, Triangulation, Visitor>

CGAL::Kinetic::Enclosing_box_2<Traits>

Definition

The class *Kinetic::Enclosing_box_2<Traits>* keeps the points in the simulation inside of a box. Whenever the points come close to the wall of the box they bounce off of the wall.

Note that, in general, points hit the wall of the box at times which are not easily represented by standard (rational) number types. The resulting trajectories would also have non-rational coefficients, complicating and slowing the simulation. In order to handle this, the *Kinetic::Enclosing_box_2<Traits>* bounces the points at the nearest easily representable time before the point would leave the box.

#include <CGAL/Kinetic/Enclosing_box_2.h>

Types

Kinetic::Enclosing_box_2<Traits>:: NT	The number type used to represent the walls of the box and perform calculations. Generally this is *Traits::NT*.

Creation

Kinetic::Enclosing_box_2<Traits> eb(Traits, NT xmin, NT xmax, NT ymin, NT ymax);

This constructs a bounding box with the dimensions specified by the last 4 arguments. They are optional and will take the values ± 10 if omitted.

CGAL::Kinetic::Enclosing_box_3<Traits>

Definition

The class *Kinetic::Enclosing_box_3<Traits>* keeps the points in the simulation inside of a box. Whenever the points come close to the wall of the box they bounce off of the wall.

Note that, in general, points hit the wall of the box at times which are not easily represented by standard (rational) number types. The resulting trajectories would also have non-rational coefficients, complicating and slowing the simulation. In order to handle this, the *Kinetic::Enclosing_box_3<Traits>* bounces the points at the nearest easily representable time before the point would leave the box.

#include <CGAL/Kinetic/Enclosing_box_3.h>

Types

Kinetic::Enclosing_box_3<Traits>:: NT	The number type used to represent the walls of the box and perform calculations. Generally this is *Traits::NT*.

Creation

Kinetic::Enclosing_box_3<Traits> eb(Traits, NT xmin, NT xmax, NT ymin, NT ymax, NT zmin, NT zmax);

This constructs a bounding box with the dimensions specified by the last 6 arguments. They are optional and will take the values ± 10 if omitted.

Class

CGAL::Kinetic::Erase_event<ActiveObjectsTable>

Definition

This event erases a point from the *ActiveObjectsTable* when it is processed.

#include <CGAL/Kinetic/Erase_event.h>

Is Model for the Concepts

Kinetic::Simulator::Event

Creation

Kinetic::Erase_event<ActiveObjectsTable> i(ActiveObjectsTable::Key k, ActiveObjectsTable::Handle t);

Erase the object k from the table t when processed.

See Also

Kinetic::ActiveObjectsTable, *Kinetic::Active_objects_vector<MovingObject>*.

Example

```
typedef CGAL::Kinetic::Exact_simulation_traits Simulation_traits;
typedef Simulation_traits::Kinetic_kernel::Point_2 Moving_point_2;
typedef
   CGAL::Kinetic::Insert_event<Simulation_traits::Active_points_2_table>
   Insert_event;
typedef CGAL::Kinetic::Delaunay_triangulation_2<Simulation_traits> KDel;

Simulation_traits tr;

KDel kdel(tr);

Moving_point_2 mp(Moving_point_2::NT(0),
                  Moving_point_2::NT(0));
tr.simulator_handle()->new_event(Simulation_traits::Simulator::Time(3),

   Erase_event(*tr.active_objects_table_handle()->keys_begin(),

   tr.active_points_2_table_handle()));
```

Kinetic::EventLogVisitor

Definition

This concept is for visitors which maintain a text log of events.

Types

Kinetic::EventLogVisitor:: Event_iterator	An iterator through strings defining the events that occurred. Each event is represented by a *std::string*.

Operations

Event_iterator	*v.events_begin()*	Begin iterating through the events.
Event_iterator	*v.events_end()*	

Has Models

Kinetic::Delaunay_triangulation_event_log_visitor_3, *Kinetic::Delaunay_triangulation_event_log_visitor_2*, *Kinetic::Regular_trianglation_event_log_visitor_3*, *Kinetic::Sort_event_log_visitor*

Class

CGAL::Kinetic::Insert_event<ActiveObjectsTable>

Definition

This event inserts a point into the *ActiveObjectsTable* when it is processed.

#include <CGAL/Kinetic/Insert_event.h>

Is Model for the Concepts

Kinetic::Simulator::Event

Creation

Kinetic::Insert_event<ActiveObjectsTable> i(ActiveObjectsTable::Data o, ActiveObjectsTable::Handle t);

Insert the object o, into the table t when processed.

See Also

Kinetic::ActiveObjectsTable, *Kinetic::Active_objects_vector<MovingObject>*.

Example

```
typedef CGAL::Kinetic::Exact_simulation_traits Simulation_traits;
typedef Simulation_traits::Kinetic_kernel::Point_2 Moving_point_2;
typedef
   CGAL::Kinetic::Insert_event<Simulation_traits::Active_points_2_table>
   Insert_event;
typedef CGAL::Kinetic::Delaunay_triangulation_2<Simulation_traits> KDel;

Simulation_traits tr;

KDel kdel(tr);

Moving_point_2 mp(Moving_point_2::Coordinate(0),
                  Moving_point_2::Coordinate(0));
tr.simulator_handle()->new_event(Simulation_traits::Simulator::Time(3),
                                 Insert_event(mp,

   tr.active_points_2_table_handle()));
```

CGAL::Kinetic::Qt_moving_points_2<Traits, QtWidget_2>

Class

Definition

The class *Kinetic::Qt_moving_points_2<Traits, QtWidget_2>* displays a set of moving points in 2D.

See Section 71.4.3 for an example using this class.

#include <CGAL/Kinetic/IO/Qt_moving_points_2.h>

Creation

Kinetic::Qt_moving_points_2<Traits, QtWidget_2> a(QtGui::Handle, Traits::Active_points_2_table::Handle)

default constructor.

See Also

Kinetic::Qt_widget_2<Simulator>.

Class

CGAL::Kinetic::Qt_triangulation_2<KineticTriangulation_2, QtWidget_2, QtMovingPoints_2>

Definition

The class draws a triangulation into a *CGAL::Qt_widget_2*. This class is very simple and a good one to look at if you want to see how to draw your own two dimensional kinetic data structure.

See Section 71.4.3 for an example using this class.

#include <CGAL/Kinetic/IO/Qt_triangulation_2.h>

Creation

Kinetic::Qt_triangulation_2<KineticTriangulation_2, QtWidget_2, QtMovingPoints_2> a(KineticTriangulation_2::Handle, QtWidget_2::Handle, QtMovingPoints_2::Handle)

Construct the object and make all the connections with the appropriate other objects.

See Also

Kinetic::Qt_widget_2<Simulator>

CGAL::Kinetic::Qt_widget_2<Simulator>

Definition

The class *Kinetic::Qt_widget_2<Simulator>* implements a graphical interface for 2D kinetic data structures.

#include <CGAL/Kinetic/IO/Qt_widget_2.h>

Types

Kinetic::Qt_widget_2<Simulator>:: Listener	The listener base to listen for when to update the picture. This class includes an extra method *Qt_widget widget()* which returns the *Qt_widget* object which can be used for drawing.

Creation

*Kinetic::Qt_widget_2<Simulator> a(int argc, char *argv[], Simulator::Handle);*

default constructor.

```
#include <CGAL/basic.h>
#include <CGAL/Kinetic/Exact_simulation_traits.h>
#include <CGAL/Kinetic/Delaunay_triangulation_2.h>
#include <CGAL/Kinetic/Delaunay_triangulation_vertex_base_2.h>
#include <CGAL/Kinetic/Delaunay_triangulation_recent_edges_visitor_2.h>
#include <CGAL/Kinetic/Enclosing_box_2.h>
#include <CGAL/Kinetic/IO/Qt_moving_points_2.h>
#include <CGAL/Kinetic/IO/Qt_triangulation_2.h>
#include <CGAL/Kinetic/IO/Qt_widget_2.h>
#include <CGAL/Kinetic/Insert_event.h>

int main(int argc, char *argv[])
{
  typedef CGAL::Kinetic::Exact_simulation_traits Traits;

 typedef CGAL::Triangulation_data_structure_2<

    CGAL::Kinetic::Delaunay_triangulation_vertex_base_2<Traits::Instantaneous_kernel>,
     CGAL::Kinetic::Delaunay_triangulation_face_base_2<Traits> > TDS;
  typedef CGAL::Delaunay_triangulation_2<Traits::Instantaneous_kernel,
    TDS > Del;
  // Color edges based on how recently they were created
  typedef
    CGAL::Kinetic::Delaunay_triangulation_recent_edges_visitor_2<Del>
    Visitor;
  typedef CGAL::Kinetic::Delaunay_triangulation_2<Traits, Visitor, Del>
    KDel;
  typedef CGAL::Kinetic::Qt_widget_2<Traits::Simulator> Qt_gui;
```

```
  typedef CGAL::Kinetic::Qt_moving_points_2<Traits, Qt_gui> Qt_mps;
  typedef CGAL::Kinetic::Qt_triangulation_2<KDel,
   Traits::Instantaneous_kernel, Qt_gui> Qt_triangulation;
  // Keeps the points confined
  typedef CGAL::Kinetic::Enclosing_box_2<Traits> Box;

  CGAL_SET_LOG_LEVEL(CGAL::Log::LOTS);

  // Run from time 0 until a really large value of time
  Traits tr(0, 10000000);
  Box::Handle box= new Box(tr);
  KDel::Handle kdel= new KDel(tr);

  Qt_gui::Handle qtsim= new Qt_gui(argc, argv, tr.simulator_handle());

  Qt_mps::Handle qtmps= new Qt_mps(qtsim, tr);
  Qt_triangulation::Handle qtdel= new Qt_triangulation(kdel,
   tr.instantaneous_kernel_object(), qtsim);
  Traits::Kinetic_kernel::Point_2 p;
  std::cin >> p;
  tr.active_points_2_table_handle()->insert(p);
  return qtsim->begin_event_loop();
}
```

CGAL::Kinetic::Regular_triangulation_3<Traits, Visitor, Triangulation>

Class

Definition

The class *Kinetic::Regular_triangulation_3<Traits, Visitor, Triangulation>* maintains a triangulation of set of moving weighted points. Its interface is the same as *Kinetic::Delaunay_triangulation_3<Traits, Visitor, Triangulation>*.

Note that the regular triangulation tracks as points are added to the *Kinetic::ActiveObjectsTable*, but not removed from it.

The optional *Triangulation* template argument must be a model of *CGAL::RegularTriangulation_3* which has *Kinetic::Regular_triangulation_cell_base_3<Traits, Base>* as a cell base and *Kinetic::Regular_triangulation_vertex_base_3<Traits, Base>* as a vertex base.

#include <CGAL/Kinetic/Regular_triangulation_3.h>

See Also

Kinetic::Delaunay_triangulation_3<Traits, Visitor, Triangulation>. *Kinetic::RegularTriangulationVisitor_3*.

Example

```
#define CGAL_CHECK_EXACTNESS
#define CGAL_CHECK_EXPENSIVE

#include <CGAL/Kinetic/Regular_triangulation_exact_simulation_traits.h>
#include <CGAL/Kinetic/Regular_triangulation_3.h>

int main()
{
    typedef CGAL::Kinetic::Regular_triangulation_exact_simulation_traits
   Traits;
    typedef CGAL::Kinetic::Regular_triangulation_3<Traits> KDel;

    Traits tr(0,100000.0);
    KDel kdel(tr);

    Traits::Simulator::Handle sp= tr.simulator_handle();

     std::ifstream in("data/weighted_points_3");
    CGAL_assertion(in);
    in >> *tr.active_points_3_table_handle();

    std::cout << *tr.active_points_3_table_handle();

    std::cout <<  *tr.active_points_3_table_handle() << std::endl;

    kdel.set_has_certificates(true);
```

```
    sp->set_current_event_number(10000);
    return EXIT_SUCCESS;
}
```

File: examples/Kinetic_data_structures/Kinetic_regular_triangulation_3.cpp

CGAL::Kinetic::Regular_triangulation_cell_base_3<Traits, Base>

Definition

This is the base class for faces used by *Kinetic::Regular_triangulation_3<Traits, Triangulation, Visitor>*.

#include <CGAL/Kinetic/Regular_triangulation_cell_base_3.h>

Class

CGAL::Kinetic::Regular_triangulation_event_log_visitor_3

Definition

The concept *Kinetic::Regular_triangulation_event_log_visitor_3* provides a model of *Kinetic::RegularTriangulationVisitor_3* and *EventLogVisitor* which logs edge flip events.

Is Model for the Concepts

Kinetic::RegularTriangulationVisitor_3, *Kinetic::EventLogVisitor*

See Also

Kinetic::Regular_triangulation_3<Traits, Triangulation, Visitor>

CGAL::Kinetic::Regular_triangulation_instantaneous_kernel< ActiveObjectsTable, StaticKernel>

Class

Definition

The class *Kinetic::Regular_triangulation_instantaneous_kernel<ActiveObjectsTable, StaticKernel>* is an instantaneous kernel for use with a regular triangulation data structure. There is not currently a reason for the user to call this directly unless the user wants created their own simulation traits as it is included as part of the *Kinetic::Regular_triangulation_exact_simulation_traits*.

#include <CGAL/Kinetic/Regular_triangulation_instantaneous_kernel.h>

Is Model for the Concepts

CGAL::RegularTriangulationTraits_3, *Kinetic::InstantaneousKernel*

See Also

Kinetic::Regular_triangulation_3<Traits, Visitor, Triangulation>.

Class

CGAL::Kinetic::Regular_triangulation_vertex_base_3<Traits, Base>

Definition

This is the base class for the vertices of the triangulation class used by *Kinetic::Regular_triangulation_3<Traits, Triangulation, Visitor>*.

#include <CGAL/Kinetic/Regular_triangulation_vertex_base_3.h>

Kinetic::RegularTriangulationVisitor_3

Concept

Definition

This concept is for proxy objects which get notified when a kinetic regular triangulation changes. It inherits all the methods of *DelaunayTriangulationVisitor_3*.

Is Model for the Concepts

Kinetic::DelaunayTriangulationVisitor_3

Operations

void *v.pre_move(Key, Cell)*

The point defined by *Key* is about to move from the cell.

void *v.post_move(Key, Cell)*

The point defined by *Key* just moved to the cell.

Has Models

Kinetic::Regular_triangulation_visitor_base_3, *Kinetic::Regular_triangulation_event_log_visitor_3*

Class

CGAL::Kinetic::Regular_triangulation_visitor_base_3

Definition

The concept *Kinetic::Regular_triangulation_visitor_base_3* provides a model of *Kinetic::RegularTriangulationVisitor_3*. You can extend this class if you only want to implement a few methods from *Kinetic::RegularTriangulationVisitor_3*.

Is Model for the Concepts

Kinetic::RegularTriangulationVisitor_3

Creation

Kinetic::Regular_triangulation_visitor_base_3 a;

default constructor.

See Also

Kinetic::Regular_triangulation_3<Traits, Triangulation, Visitor>

CGAL::Kinetic::Sort_event_log_visitor

Class

Definition

The concept *Kinetic::Sort_event_log_visitor* provides a model of *SortVisitor* and *EventLogVisitor* which logs changes to the structure.

Is Model for the Concepts

Kinetic::SortVisitor, *Kinetic::EventLogVisitor*

See Also

Kinetic::Sort<Traits, Visitor>

CGAL::Kinetic::Sort<Traits, Visitor>

Definition

The class *Kinetic::Sort<Traits, Visitor>* maintains a sorted list of objects. It is the simplest kinetic data structure provided and is a good place to start when looking at the basics of implementing a kinetic data structure.

The *Kinetic::SortVisitor* can be used to monitor what is happening.

#include <CGAL/Kinetic/Sort.h>

Creation

Kinetic::Sort<Traits, Visitor> s(Traits tr);	The basic constructor.

Types

Kinetic::Sort<Traits, Visitor>:: Visitor	The type of the visitor.
Kinetic::Sort<Traits, Visitor>:: Traits	The traits type.
Kinetic::Sort<Traits, Visitor>:: Vertex_handle	The handle used to refer to vertex in the sorted list. Dereferencing this returns a *Key* into the *ActiveObjectsTable*.
Kinetic::Sort<Traits, Visitor>:: Handle	A reference counted pointer to be used for storing references to the object.
Kinetic::Sort<Traits, Visitor>:: Const_handle	A reference counted pointer to be used for storing references to the object.

Operations

Visitor&	*s.visitor()*	Access the visitor.
Traits&	*s.traits()*	Access the traits.
Vertex_handle	*s.insert(Point_key k)*	Insert the point.
void	*s.erase(Vertex_handle k)*	Erase the point.
Iterator	*s.begin()*	Begin iterating through the ordered *Vertex_handle*s (the iterator is convertible to *Vertex_handle*.
Iterator	*s.end()*	End iterating through the ordered *Vertex_handle*s (the iterator is convertible to *Vertex_handle*.

See Also

Kinetic::Ref_counted<T>

Class

CGAL::Kinetic::Sort_visitor_base

Definition

The concept *Kinetic::Sort_visitor_base* provides a model of *Kinetic::SortVisitor*. You can extend this class if you only want to implement a few methods from *Kinetic::SortVisitor*.

Is Model for the Concepts

Kinetic::SortVisitor.

Creation

Kinetic::Sort_visitor_base a;	default constructor.

See Also

Kinetic::Sort<Traits, Visitor>

Kinetic::SortVisitor

Concept

Definition

This concept is for proxy objects which have functions called on them when a *Kinetic::Sort<Traits, Visitor>*.

Operations

void	*v.pre_remove_vertex(Vertex_handle)*	
		The vertex is about to be deleted.
void	*v.post_remove_vertex(Point_key)*	
		The vertex was just removed.
void	*v.pre_insert_vertex(Point_key)*	
		The vertex is about to be inserted into the cell (although the *Cell_handle* might be *NULL*).
void	*v.post_insert_vertex(Vertex_handle)*	
		The vertex was just inserted.
void	*v.change_vertex(Vertex_handle)*	
		Something changed at the vertex.
void	*v.pre_swap(Vertex_handle, Vertex_handle)*	
		The pair of vertices is about to be exchanged.
void	*v.post_swap(Vertex_handle, Vertex_handle)*	
		The pair of vertices was just swapped.

Has Models

Kinetic::Sort_visitor_base, *Kinetic::Sort_event_log_visitor*

Chapter 72

Kinetic Framework

Daniel Russel

Contents

This chapter describes a framework for implementing kinetic data structures and sweepline algorithms. If you just would like to use existing kinetic data structures, please read Chapter 71 instead. Readers wishing to brush up on their familiarity with kinetic data structures or better understand the terminology we use should read Section 71.2 of that chapter. A brief overview of the framework can be found in Section 71.3 (also of that chapter) and it too is recommended reading. Here we dive right in to discussing to discussing the architecture of the framework in Section 72.1 and finally we give several examples of using the framework to implement a kinetic data structure in Section 72.3. The framework makes heavy use of our *Polynomial_kernel* package to provide models of the *Kinetic::FunctionKernel* concept.

The framework was first presented at ALENEX [GKR04].

72.1 Architecture

This package provides a framework to allow exact implementation of kinetic data structures and sweepline algorithms. Below we discuss in detail each one of the first four major concepts which help in implementing

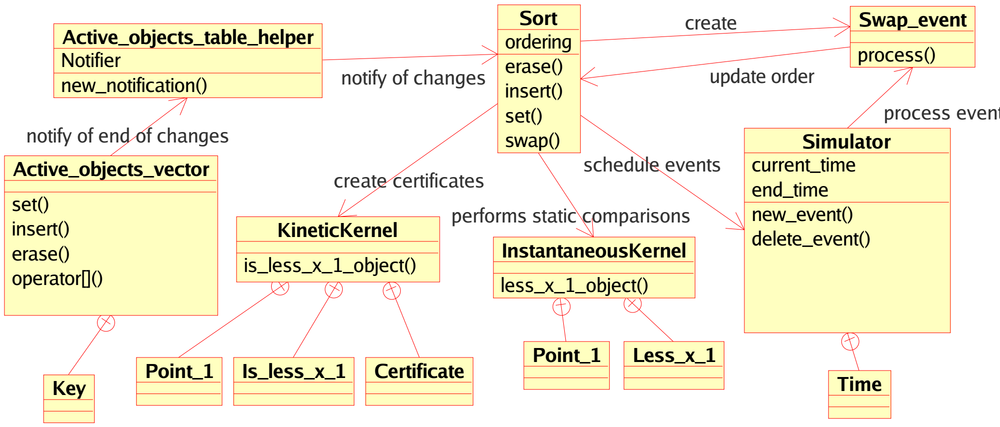


Figure 72.1: The figure, identical to the one in the overview of the previous chapter, shows the interaction between the *Kinetic::Sort<Traits, Visitor>* kinetic data structure and the various pieces of our framework. Other, more complicated, kinetic data structures will also use the *Kinetic::InstantaneousKernel* in order to insert/remove geometric primitives and audit themselves. *Kinetic::Sort<Traits, Visitor>* uses the sorting functionality in STL instead.

kinetic data structures: the *Kinetic::Simulator*, the *Kinetic::Kernel*, the *Kinetic::ActiveObjectsTable* and the *Kinetic::InstantaneousKernel*. The *Kinetic::FunctionKernel* concept is discussed separately in Section 72.2.

72.1.1 The Kinetic::Simulator

The *Kinetic::Simulator* is the central repository of all active events. It maintains the event queue and can use its knowledge of the events in the queue to find times for the kinetic data structures to easily check their own correctness (this will be discussed in more detail later in this section). Kinetic data structures call methods of the *Kinetic::Simulator* to schedule new events, deschedule old ones and access and change data contained in already scheduled events (the operations on existing events are performed using a key which was returned when the event was scheduled). For controlling the simulation, methods in the *Kinetic::Simulator* allow stepping through events, advancing time and even running the simulation backwards (that is we run the simulation with the time running in the opposite direction).

The kinetic sorting example in Figure 71.4.1 shows the basic usage of the *Kinetic::Simulator*. First, the *Simulator* is created by the *Kinetic::SimulationTraits*. The kinetic data structure gets a handle to the simulator from the traits class and uses the handle to add its events to the simulation. The *Kinetic::Simulator* is then told to advance time up until the end of the simulation, processing all events along the way.

Each event is represented by a *Kinetic::Simulator::Time* and an instance of a model of the *Kinetic::Simulator::Event* concept. Models of the *Kinetic::Simulator::Event* concept are responsible for taking the appropriate action in order to handle the kinetic event they represent. Specifically, the *Kinetic::Simulator::Event* concept specifies one method, *Kinetic::Simulator::Event::process()*, that is called when the event occurs. The body of the *Kinetic::Simulator::Event::process()* method typically simply calls a method of the kinetic data structure that created the event; for example in our kinetic sorting example, processing an event means calling the *Kinetic::Sort<Traits, Visitor>::swap(Iterator)* method of the kinetic sorting data structure.

In the model of the *Kinetic::Simulator* concept that we provide, *Kinetic:Default_simulator<FunctionKernel, EventQueue>*, any model of the *Kinetic::Simulator::Event* concept can be inserted as an event. This ability implies that events can be mixed at run time, which is essential when we want to support multiple kinetic data structures operating on the same set of moving geometric primitives.

The *Kinetic::Simulator::Time* concept is defined by the simulator, typically to be some representation of a root of a polynomial, taken from the *Kinetic::FunctionKernel* (details of the algebraic side of the package will be discussed in Section 72.2). For most kinetic data structures *Kinetic::Simulator::Time* only needs to support comparisons (we need to compare events, in order to process them in the correct order) and a few other non-arithmetic operations.

When the failure times of certificates are sorted exactly (as opposed to when we numerically approximate the roots of the certificate polynomials) the correctness of kinetic data structures can be easily verified. Let I be an open interval between the last event processed and the next event to be processed. As was mentioned in the introduction kinetic data structures do not change combinatorially in I. In addition, although the static data structures can be degenerate at the roots defining the two ends of the interval, they are not, in general, degenerate in the interior. An independent check of the integrity of kinetic data structures can be provided by, for example, using an *Kinetic::InstantaneousKernel* (cf. Subsection 72.1.4) to rebuild the static version of the structure from scratch at some time interior to I and compare it to the kinetic version. This auditing can typically catch algorithmic or programming errors much closer to the time they arise in the simulation than, for example, using visual inspection. Such easy auditing is one of the powerful advantages of having an exact computational framework since, as with static data structures, when using inexact computations differentiating between errors of implementation and numeric errors is quite tricky.

Kinetic data structures receive alerts of appropriate times to audit themselves using a notification framework. The same framework is also used by the *Kinetic::ActiveObjectsTable* to alert kinetic data structures when the set of primitives changes (see Subsection 72.1.3). To use the notification framework, the kinetic data structure creates a proxy object which implements a standard *Listener* interface. It then registers this proxy with the *Kinetic::Simulator*. When the *Kinetic::Simulator* finds an appropriate time for the kinetic data structures to audit themselves it calls the function *Listener::new_notification(Type)* on each of the registered proxy objects. A helper for creating such proxy objects, called *Kinetic::Simulator_kds_listener<Listener, KDS>*, is provided by the framework. It translates the notification into a function call (*audit()*) on the kinetic data structure. Pointers in the notification framework are reference counted appropriately to avoid issues caused by the creation and destruction order of kinetic data structures and the simulator. See Section 72.1.5 for a more complete discussion of this part of the framework.

Internally the *Kinetic::Simulator* maintains a priority queue containing the scheduled events. The type of the priority queue is a template argument to our *Kinetic::Simulator* model and, as such, it can be replaced by the user. In our package, we provide two different types of priority queues, a heap and a two-list priority queue. A two-list queue is a queue in which there is a sorted front list, containing all events before some time and an unsorted back list. The queue tries to maintain a small number of elements in the front list, leaving most of them in the unsorted main pool. The two-list queue, although an unconventional choice, is our default queue when using exact computation because it minimizes comparisons involving events that are far in the future. These events are likely to be deleted before they are processed, so extra work done structuring them is wasted. Our experiments have shown that, for example, the two-list queue causes a 20% reduction in running time relative to a binary heap for Delaunay triangulations with degree 3 polynomial motions and 20 points.

72.1.2 The Kinetic::Kernel

The *Kinetic::Kernel* is structured very much like static CGAL kernels. It defines a number of primitives, which in the model provided are *Kinetic::Kernel::Point_1*, *Kinetic::Kernel::Point_2*, *Kinetic::Kernel::Point_3* and *Kinetic::Kernel::Weighted_point_3*. The primitives are defined by a set of Cartesian coordinates each of which is a function of time, a *Kernel::MotionFunction*. In addition it defines constructions and certificate generators which act on the primitives. The certificate generators are the direct analog of the non-kinetic predicates. Each certificate generator take a number of primitives as arguments, but instead of producing an element from a discrete set they produce a set of discrete failure times for the certificate. These failure times are wrapped in a model of *Kinetic::Certificate*.

A *Kinetic::Certificate* is a simple object whose primary function is to produce a *Kinetic::Simulator::Time* object representing the failure time of the certificate. Since, the handling of most certificate failures involves creating a new certificate whose certificate function is the negation of the old certificate function, a *Kinetic::Certificate* object caches any work that could be useful to isolate future roots of the certificate function (such as the Sturm sequence of the certificate function). To illustrate this further, if you have two one-dimensional points with coordinate functions $p_0(t)$ and $p_1(t)$, the certificate that the first moving point is after the second corresponds to the inequality $p_0(t) - p_1(t) > 0$. When the certificate fails and the two points cross, the new certificate is $p_1(t) - p_0(t) > 0$, which is the negated version of the certificate just processed and which has the same roots.

The model of *Kinetic::Kernel* provided includes the certificate generators necessary for Delaunay triangulations (in one, two and three dimensions) and regular triangulations (in 3D). New certificates can be fairly easily added. An example is included in the distributed code.

72.1.3 The Kinetic::ActiveObjectsTable

The *Kinetic::ActiveObjectsTable* stores a set of kinetic primitives. Its purpose is to notify kinetic data structures when new primitives are added, when primitives are removed or when a trajectories change. Each primitive is uniquely identified by a Key, assigned by the table when the primitive is added, that can be used to change or remove it. We provide one model of the *Kinetic::ActiveObjectsTable* concept, called *Kinetic::Active_objects_vector<MovingObject>* which stores all the moving primitives in an *std::vector<D>*.

Notifications of changes to the set of active objects are handled using a setup similar to the *Kinetic::Simulator* audit time notification. We provide a helper class, *Kinetic::Active_objects_listener_helper<ActiveObjectsTable, KDS>*, which translates the notifications into *insert(Key)*, *erase(Key)* or *set(Key)* function calls on the kinetic data structure.

72.1.4 The Kinetic::InstantaneousKernel

The *Kinetic::InstantaneousKernel* allows existing CGAL data structures to be used on moving data as it appears at some instant of time. Models of this concept are, by definition, models of a CGAL Kernel or a traits class, and, therefore, can then be used as the traits class of CGAL's algorithms and data structures.

Consider for example the kinetic Delaunay data structure in either two or three dimensions. Internally, it uses a *Delaunay_triangulation_2<Traits, Tds>* or *Delaunay_triangulation_3<Traits, Tds>* to represent the triangulation, instantiated with a model of the *Kinetic::InstantaneousKernel* concept as its traits class. At initialization, as well as at times during the simulation when we want to insert a point to the kinetic Delaunay triangulation, a static version of the Delaunay triangulation is conceptually instantiated. More precisely, the time for the copy of the model of the *Kinetic::InstantaneousKernel* stored in the CGAL triangulation is set to be the current time (or rather, as discussed in the introduction, a more convenient time determined by the *Kinetic::Simulator* combinatorially equivalent to the current time). The kinetic data structure then calls the *Delaunay_triangulation_3<Traits, Tds>::insert(Point)* insert method to insert the point. The static insert method called uses various predicate functors on the moving points which evaluate to the values that the predicates have at that instant in time. Removal is handled in an analogous manner. Auditing of the geometric structure is easily handled in a similar manner (in the case of Delaunay triangulations by simply calling the *verify()* method after setting the time).

72.1.5 Miscellaneous: notification and reference management

We describe some coding conventions used, graphical display, notification and reference management support in the framework in the following sections.

Reference management

A number of objects need to maintain pointers to other independent objects. For example, each kinetic data structure must have access to the *Kinetic::Simulator* so that it can schedule and deschedule events. These pointers are all reference counted in order to guarantee that they are always valid. We provide a standard reference counting pointer and object base to facilitate this, namely *Ref_counted<Object>*.

Each shared object in the framework defines a type *Handle* which is the type for a reference counter pointer pointing to it. These should be used for storing pointers to the objects in order to avoid dangling pointers. In addition, many of the objects expect such pointers as arguments.

Runtime event passing

Runtime events must be passed from *notifiers*, namely the *Kinetic::ActiveObjectsTable* and the *Kinetic::Simulator* to *listeners*, typically the kinetic data structures. For example, kinetic data structures are notified when new primitives are added to the *Kinetic::ActiveObjectsTable*. On reciving the notification, it will add the new primitive to the combinatorial structure it is maintaining. The events are passed using a simple, standardized notification interface. To receive notifications, the listener first defines a small proxy class which inherits from a *Listener* base type provided by the notifier. On creation, the *Listener* base class registers itself with the notifier on construction (and unregisters itself on destruction).

When the some state of the notifier changes, it calls the *new_notification* method on the listener proxy object provided and passes it a label corresponding to the name of the field that changed. The proxy object can then call an appropriate method on the kinetic data structure or whetever the listening class is.

In order to unregister on destruction, the *Listener* must store a (reference counted) pointer to the object providing notifications. This pointer can be accessed through the *notifier()* field. The *Listener* object stores a reference counted pointer to the notifying object, while the notifying object stores a plain pointer to the *Listener*. It can do this since the *Listener* is guaranteed to unregister itself when it is destroyed. This avoids circular reference counted pointers as well as dangling pointers.

72.2 Algebraic Kernel

The interface between the algebraic kernel and the kinetic data structures package was kept quite minimal in order to ease the implementation of various underlying computation models. The interface is detailed in the reference page (*Kinetic::FunctionKernel*).

We provide models of the algebraic kernel that handle polynomial *Kinetic::Function* objects. The provided models perform

- exact computations using Sturm sequences to isolate roots
- exact computations using Descartes rule of sign in order to isolate roots (Sturm sequences are also used in order to properly handle even multiplicity roots)

- filtered exact computations using Descartes rule of sign
- numeric (inexact) root approximations
- numeric root approximations which take advantage of certain assumptions that can be made about the types of polynomials solved in the process of evaluating kinetic data structures
- a wrapper for CORE::Expr which implements the required concepts.

The exact models, which we implement the numerics for, handle non-square-free polynomials and polynomials with arbitrary field number type coefficients and are quite robust.

72.2.1 Kinetic::FunctionKernel customized for kinetic data structures

There are several modifications we can make to how the roots are handled to optimize for the case of kinetic data structures. The first are motivated by the question of how to handle degeneracies (certificate functions which have roots at the same time). Naively, there is no way to differentiate between a certificate which fails immediately when it is created and one whose function is momentarily 0, but will be valid immediately in the future. In order to handle such degeneracies we ensure that all the certificate function generators produce certificate functions which are positive when the corresponding certificates are valid. Then, if we have a degeneracy we can differentiate between a certificate which fails immediately and one which is simply degenerate by looking at the sign of the certificate function immediately following the root (equivalently, by looking at the derivative). In addition, this allows us, under the assumption that computations are performed exactly, to check that all certificates are not invalid upon creation.

The assumption that certificates are positive when valid is particular useful when using numeric solvers. Without it there is no reliable way to tell whether a root near the current time is the certificate having become valid just before the current time, or failing shortly in the future. Testing the sign of the function immediately after the root reliably disambiguates the two cases.

In addition, we have to specially handle even roots of functions. For the most part these can just be discarded as dropping an even root is equivalent to perturbing the simulation to remove the degeneracy. However, when we are using the *Kinetic::Simulator* to audit the kinetic data structures, they most be broken up in to two, equal, roots to avoid auditing at the degeneracy.

72.3 Examples

We provide a number of examples of different levels of usage of the kinetic data structures framework, both for kinetic data structures as well as sweepline algorithms.

To see how to use existing kinetic data structures, look at the examples in the previous chapter such as Section 71.4.1.

The here we cover implementing kinetic data structures. The examples explained are

- A trivial kinetic data structure which has all the parts of a full kinetic data structure but doesn't do much in Section 72.3.2.
- Adding a new type of certificate to a kernel in Section 72.3.3.

In order to see more detail about how to implement a kinetic data structure, the best place to start is the source code for the kinetic sorting data structure, *Kinetic::Sort<Traits, Visitor>*. Once you are familiar with that, *Kinetic::Delaunay_2<Traits, Triangulation, Visitor>* is the next step in complexity.

We will first explain in detail how a typical kinetic data structure uses the various pieces of the framework, then move on to showing the actual code for a simpler data structure.

72.3.1 Using the Pieces of the Package

Here we will explain how the kinetic sorting data structure uses the various pieces of the package. A schematic of its relationship to the various components is shown in the UML diagram in Figure 72.1. In this subsection we abuse, for reasons of simplicity of presentation, the concept/model semantics: when we refer to concepts we actually refer to an instance of a model of them.

As with most kinetic data structures, *Kinetic::Sort<Traits, Visitor>* maintains some sort of combinatorial structure (in this case a sorted doubly linked list), each element of which has a corresponding certificate in the event queue maintained by the simulator. In the case of sorting, there is one certificate maintained for each "edge" between two consecutive elements in the list.

On creation, the data structure is passed a copy of the *Kinetic::SimulationTraits* for this simulation, which it saves for future use. It gets a handle to to the *Kinetic::ActiveObjectsTable* by calling the *Kinetic::SimulationTraits::active_points_1_table_handle()* method and registers a proxy with the table in order to receive notifications of changes to the point set. The *Kinetic::SimulationTraits* method returns a handle to, rather than a copy of, the *Kinetic::ActiveObjectsTable*, since the table must be shared between all the kinetic data structures using these points. The handles are reference counted pointers, thus saving the user from worrying about cleaning things up properly.

When new points are added to the model of the *Kinetic::ActiveObjectsTable*, the table calls the *new_notification()* method on the proxy of the kinetic data structure, which in turn calls the *insert(Point_key)* method of the kinetic data structure. The *Point_key* here is the key which uniquely identifies the newly inserted point in the table. The data structure then requests an instance of a model of the *Kinetic::InstantaneousKernel* from the *Kinetic::SimulationTraits*. It sets the time on the instantaneous kernel to the time value gotten from the *Kinetic::Simulator::current_time_nt()* method. This method returns a field number type that is between the previous and next event, as discussed in the introduction. An instance of the *Kinetic::InstantaneousKernel::Compare_x_1* predicate (wrapped in order to make it return *less*) and the STL function *std::upper_bound()* are then used to insert the new point in the sorted list. For each inserted object, the kinetic data structure removes the no longer relevant certificate from the event queue by calling the *Kinetic::Simulator::delete_event(Key)* function and creates two new certificates using a *Kinetic::Kernel::Compare_x_1* certificate functor. The new certificates are inserted in the event queue by calling the *Kinetic::Simulator::new_event(Time, Event)* method where *Kinetic::Simulator::Event* is a proxy object which instructs the sort kinetic data structure to swap two points when its *process()* method is called.

Now that the kinetic data structure has been initialized, the simulator is instructed to process all events. Each time an event occurs, the simulator calls the *process()* method on the corresponding proxy object. The proxy, in turn, tells the sort kinetic data structure to swap the two points whose order has changed.

The *Kinetic::Simulator* can periodically instruct the kinetic data structures to audit themselves. As is explained in Section 72.1.1, a proxy object maps the notification on to an *audit()* function call in the kinetic data structure. To audit itself the kinetic data structure builds a list of all the current points and uses *std::sort* to sort this list using a comparison function gotten from the *Kinetic::InstantaneousKernel*. This sorted list is compared to the maintained one to verify correctness. This auditing could also have been done by evaluating the *Kinetic::InstantaneousKernel* predicate for each sorted pair. Since auditing a kinetic data structure typically requires at least linear time in the size of the combinatorial structure, the auditing procedure in between events is deactivated by default. The user can however easily switch it on by defining the *CGAL_CHECK_EXACTNESS* and *CGAL_CHECK_EXPENSIVE* CGAL macros.

This general structure of the interaction between the kinetic data structure and the framework is shared by all of the provided kinetic data structures and has proved itself to go quite far.

72.3.2 The Trivial Kinetic Data Structure

To show how to implement such things, instead of presenting a full kinetic data structure, we present a trivial one which maintains one event in the queue which maintains one event in the queue, scheduled to occur one time unit after the last change was made to the set of active primitives. Two classes are defined, the *Trivial_event*, and the *Trivial_kds*. The event classes must be declared outside of the kinetic data structure so that the *operator<<* can be defined for them.

The kinetic data structure maintains the invariant that it was one event in the queue at all times. This event ccurs one time unit after the last event or change in the set of objects occurs. As a result, the kinetic data structure has the main parts of a real one–it responds to changes in trajectories of the objects and certificate failures (when the event expires).

The public methods can be grouped into three sets which are shared with almost all other kinetic data structures:

- *has_certificates* and *set_has_certificates* which checks/sets whether the kinetic data structure is currently maintaining certificates.
- *insert*, *set*, *erase* which are called by the *Kinetic::Active_objects_listener_helper* in response to the addition, modification, or deletion of an object to, in or from the simulation.
- *audit* which is called periodically by the *Kinetic::Simulator_kds_listener* when kinetic data structures can easily audit themselves.

In addition, it has one method which is called when a certificate fails. The name/existence of such methods depend on the nature of the kinetic data structure in question.

Like many kinetic data structures, it takes a *Kinetic::SimulationTraits* as a template argument. This traits class defines the types needed for the simulation and is responsible for instantiating them.

```
#include <CGAL/Kinetic/Ref_counted.h>
#include <CGAL/Kinetic/Exact_simulation_traits.h>
#include <CGAL/Kinetic/Active_objects_listener_helper.h>
#include <CGAL/Kinetic/Simulator_kds_listener.h>
...

// This must be external since operator<< has to be defined
template <class KDS>
struct Trivial_event
{
  Trivial_event(){}
  Trivial_event(KDS* kds): kds_(kds) {
  }
  void process() const
  {
    kds_->process();
  }
```

```
  KDS* kds_;
};

template <class KDS>
std::ostream &operator<<(std::ostream &out,
                         const Trivial_event<KDS> &) {
  out << "\"An event\"";
  return out;
}


template <class Traits>
struct Trivial_kds: CGAL::Kinetic::Ref_counted<Trivial_kds<Traits> >
{
  typedef Trivial_kds<Traits> This;
  typedef typename Traits::Active_points_1_table::Data Point;
  typedef typename Traits::Simulator::Time Time;
  typedef typename Traits::Active_objects_table::Key Point_key;
  typedef typename Traits::Simulator::Event_key Event_key;
  typedef CGAL::Kinetic::Active_objects_listener_helper<
    typename Traits::Active_points_1_table::Listener, This>
   Active_objects_helper;
  typedef CGAL::Kinetic::Simulator_kds_listener<
    typename Traits::Simulator::Listener, This> Simulator_helper;

  typedef Trivial_event<This> Event;

  Trivial_kds(Traits tr): has_certificates_(true),
                          tr_(tr),
                          nth_(tr.active_points_1_table_handle(), this),
                          sh_(tr.simulator_handle(), this){}

  // this method is called with the value true when the event is
   processed
  void process(bool tf) {
     event_= Event_key();
     set_has_certificates(false);
     set_has_certificates(true);
  }

  void audit() const
  {
     ...
  }

  void set_has_certificates(bool tf) {
    typename Traits::Simulator::Handle sp= tr_.simulator_handle();
    if (has_certificates_ != tf) {
      has_certificates_=tf;
      if (has_certificates_) {
        bool ev= event_;
        CGAL_assertion(!ev);
        Time t= CGAL::to_interval(sp->current_time()).second+1;
        event_= sp->new_event(t, Event(this));
```

```
      } else if (event_) {
        sp->delete_event(event_);
        event_=Event_key();
      }
    }
  }

  bool has_certificates() const {
    return has_certificates_;
  }

  void insert(Point_key k) {
    if (has_certificates_) {
      set_has_certificates(false);
      set_has_certificates(true);
    }
  }

  void set(Point_key k) {
    if (has_certificates_) {
      set_has_certificates(false);
      set_has_certificates(true);
    }
  }

  void erase(Point_key k) {
    if (has_certificates_) {
      set_has_certificates(false);
      set_has_certificates(true);
    }
  }

  ~Trivial_kds(){
     set_has_certificates(false);
  }

protected:
  bool has_certificates_;
  Event_key event_;
  Traits tr_;
  Active_objects_helper nth_;
  Simulator_helper sh_;
};
```

72.3.3 Adding a New Certificate Type

The following example shows how to add a new type of certificate to a simulation.

First we code the actual certificate function generator. It must take some sort (or sorts) of kinetic primitives, compute some function from their coordinates.

```
template <class KineticKernel>
```

```
struct Positive_x_f_2 {
  typedef typename KineticKernel::Certificate_function result_type;
  typedef typename KineticKernel::Point_2 argument_type;
  result_type operator()(const argument_type &p){
    return result_type(p.x()- result_type(0));
  }
};
```

Then we define a kinetic kernel which includes this predicate. To do this we wrap the function generator generator in a *Kinetic::Certificate_generator<Kernel, Generator>*. This wrapper uses the generator to create the certificate function and then the *Kinetic::FunctionKernel* to solve the certificate function. The result is wrapped in a *Kinetic::Certificate* object.

```
template <class FunctionKernel>
class My_kinetic_kernel:
  public CGAL::Kinetic::Cartesian<FunctionKernel> {
  typedef CGAL::Kinetic::Cartesian<FunctionKernel> P;
  typedef My_kinetic_kernel<FunctionKernel> This;
public:
  typedef CGAL::Kinetic::internal::Certificate_generator<This,
   Positive_x_f_2<This> > Positive_x_2;
  Positive_x_2 positive_x_2_object() const
  {
    return Positive_x_2(P::function_kernel_object());
  }
};
```

Now we have the unfortunately rather messy part of assembling a new *Kinetic::SimulationTraits* model. This is done in two steps for convenience.

```
struct My_simulation_traits {
  typedef My_simulation_traits This;

  typedef CGAL::Exact_predicates_exact_constructions_kernel
   Static_kernel;
  //typedef
   CGAL::Regular_triangulation_euclidean_traits_3<Static_kernel_base>
   Static_kernel;
  typedef CGAL::POLYNOMIAL::Polynomial<Static_kernel::FT> Function;
  typedef CGAL::POLYNOMIAL::Sturm_root_stack_traits<Function>
   Root_stack_traits;
  typedef CGAL::POLYNOMIAL::Sturm_root_stack<Root_stack_traits>
   Root_stack;
  typedef CGAL::POLYNOMIAL::Kernel<Function, Root_stack> Function_kernel;

  typedef
   CGAL::Kinetic::Handle_degeneracy_function_kernel<Function_kernel,
   false>  Simulator_function_kernel_base;
  struct Simulator_function_kernel: public
   Simulator_function_kernel_base{};

  typedef My_kinetic_kernel<Simulator_function_kernel> Kinetic_kernel;
  typedef CGAL::Kinetic::Two_list_pointer_event_queue<Function_kernel>
   Event_queue;
```

```
  typedef CGAL::Kinetic::Default_simulator<Simulator_function_kernel,
    Event_queue > Simulator;

  typedef CGAL::Kinetic::Active_objects_vector<Kinetic_kernel::Point_1>
    Active_points_1_table;
  typedef CGAL::Kinetic::Active_objects_vector<Kinetic_kernel::Point_2>
    Active_points_2_table;
  typedef CGAL::Kinetic::Active_objects_vector<Kinetic_kernel::Point_3>
    Active_points_3_table;
  // typedef Active_objects_vector<Kinetic_kernel::Weighted_point_3>
    Active_weighted_points_3_table;

  typedef CGAL::Kinetic::Default_instantaneous_kernel<This>
    Instantaneous_kernel;

  Active_points_1_table* active_points_1_table_handle() const { return
    ap1_.get();}
  Active_points_2_table* active_points_2_table_handle() const {return
    ap2_.get();}
  Active_points_3_table* active_points_3_table_handle() const {return
    ap3_.get();}
  //Active_weighted_points_3_table*
    active_weighted_points_3_table_handle() const {return awp3_.get();}

  Simulator* simulator_handle() const { return sim_.get();}
  const Static_kernel& static_kernel_object() const {return k_;}
  const Kinetic_kernel& kinetic_kernel_object() const {return kk_;}

  Instantaneous_kernel instantaneous_kernel_object() const {
    return Instantaneous_kernel(*this);
  }

  My_simulation_traits(const Simulator::Time &lb,
                       const Simulator::Time &ub): sim_(new Simulator(lb,
     ub)),
                                                   ap1_(new
    Active_points_1_table()),
                                                   ap2_(new
    Active_points_2_table()),
                                                   ap3_(new
    Active_points_3_table())
  {}


  bool is_exact() const {
    return true;
  }
protected:
  Simulator::Handle sim_;
  Active_points_1_table::Handle ap1_;
  Active_points_2_table::Handle ap2_;
  Active_points_3_table::Handle ap3_;
  //Active_weighted_points_3_table::Handle awp3_;
  Static_kernel k_;
```

```
  Kinetic_kernel kk_;
  Function_kernel fk_;
};
```

Now the simulation traits can be used by a kinetic data structure. Note that we define active point table for all dimensions. This is needed by the *Kinetic::InstantaneousKernel*, even if they are not used.

Kinetic Framework Reference Manual

Daniel Russel

72.4 Classified Reference Pages

Definition

Kinetic data structures are a way of adding motion to classical geometric data structures. CGAL provides a number of classes to aid implementation of kinetic data structures.

There are three levels at which the user can interact with the package. The user can use an existing kinetic data structure, write a new kinetic data structure, or replace parts of the framework.The first level is covered in the Chapter 72.

Main Support Classes and Concepts

Here we list the main classes and concepts provided by the framework to support implementing kinetic data structures

Other Concepts

Other Classes

72.5 Alphabetical List of Reference Pages

CGAL::Kinetic::Active_objects_listener<ActiveObjectsTable, KDS>

Class

Definition

The class *Kinetic::Active_objects_listener<ActiveObjectsTable, KDS>* acts as an intermediate between a moving object table and a KDS. It translates the *ActiveObjectsTable::Listener::IS_EDITING* notification events into appropriate calls to *KDS::insert(Key)*, *KDS::set(Key)*, *KDS::erase(Key)*.

Kinetic data structures can still take advantage of the batch editing if they are careful. The methods (such as *KDS::set(Key)* are called in lexicographical order in the *Key*s. So, when a KDS is preparing to update some certificate involving a recently set object, it can first check if the certificate involves another changed object which is lexicographically prior. If so, then the certificate has already been updated and can be skipped.

#include <CGAL/Kinetic/listeners.h>

See Also

Kinetic::Active_objects_vector<MovingObject>, *Kinetic::ActiveObjectsTable*.

Concept

Kinetic::ActiveObjectsTable

Definition

This container holds a set of objects of a particular type. It creates notifications using the standard *Multi_listener<Interface>* interface when a primitive changes or is added or deleted. Objects which are listening for events can then ask which primitives changed.

For speed, modifications to the Kinetic::ActiveObjectsTable can be grouped into editing sessions. A session is begun by calling *set_is_editing(true)* and ended by calling *set_is_editing(false)*. There is one type of notification, namely, *Listener::IS_EDITING* which occurs when the editing mode is set to false, signaling that a batch of changes is completed.

As an convenience, the change methods can be called without setting the editing state to true, this acts as if it were set to true for that one function call.

Types

Kinetic::ActiveObjectsTable:: Key A key identifying an object in the table.

Kinetic::ActiveObjectsTable:: Data The type being stored in the table.

Kinetic::ActiveObjectsTable:: Listener The base class to derive from for listening for runtime events.

The following types are iterators. Each type, *Foo_iterator* has two corresponding methods *foo_begin* and *foo_end* which allow you to iterate through the objects in the set *Foo*.

Kinetic::ActiveObjectsTable:: Key_iterator An iterator through all the valid keys in the table.

Kinetic::ActiveObjectsTable:: Changed_iterator

An iterator through all the objects which have been changed in the last editing session. The iterator iterates through the objects in lexicographical order.

Kinetic::ActiveObjectsTable:: Inserted_iterator

An iterator through all the objects which were added in the last editing session.

Kinetic::ActiveObjectsTable:: Erased_iterator An iterator through all the objects which were deleted in the last editing session.

Creation

Operations

Data	*mot*[*Key key*]	Access the object referenced by the key.
Data	*mot.at(Key key) const*	Access the object referenced by the key.
void	*mot.set_is_editing(bool is_editing)*	Set the editing state of the object. A notification is sent when the editing state is set to false after it has been true, i.e. the editing session is finished. This allows changes to be batched together.
bool	*mot.is_editing() const*	Access the editing state.
void	*mot.set(Key key, Data object)*	This method changes the motion of one moving object. The position at the current time should not be different from the previous current position. However, at the moment I do not check this as there is no reference to time in the Kinetic::ActiveObjectsTable. If *is_editing()* is not true, then it is as if the calls *set_is_editing(true)*, *set(key, value)* and finally *set_is_editing(false)* were made. If it is true, then no notifications are created.
Key	*mot.insert_object(Data ob)*	Insert a new object into the table and return a *Key* which can be used to refer to it. See *set(Key, Data)* for a description of editing modes.
void	*mot.erase(Key key)*	Delete an object from the table. The object with Key key must already be in the table. This does not necessarily decrease the amount of storage used at all. In fact, it is unlikely to do so. See *set(Key,Data)* for an explainating of how the editing modes are used.
void	*mot.clear()*	Remove all objects from the table and free all storage.
void	*mot.is_set(Key) const*	Returns true if the object has been set in the currently notified editing session. Note that this method can only be saftely called when processing an *IS_EDITING* notification.
void	*mot.is_new(Key) const*	Returns true if the object has been set in the currently notified editing session.

void	*mot.clear(Key) const*	Returns true if the object has been set in the currently notified editing session.

Has Models

Kinetic::Active_objects_vector<MovingObject>

See Also

Multi_listener<Interface>, *Kinetic::Active_objects_listener_helper<ActiveObjectsTable, KDS>*

CGAL::Kinetic::Active_objects_vector<MovingObject>

Class

Definition

MovingObjects are stored in a vector. This means that access is constant time, but storage is not generally freed. The only way to be sure is to remove all reference counts for the table or to call *clear()*.

#include <CGAL/Kinetic/Active_objects_vector.h>

Is Model for the Concepts

Kinetic::ActiveObjectsTable

Class

CGAL::Kinetic::Cartesian<FunctionKernel>

Definition

This class provides a model of *Kinetic::Kernel* for use with general Cartesian geometry.

The IO format for points is currently p_0, p_1, ... w. p_i and w are instances of Function. There IO format is typically $c_0 + c_1 t + c_2 t^2 + \ldots$. Beware of issues with CGAL IO of the coeffients as exact number typles often require that the coefficents be expressed as a/b even when b is 1.

#include <CGAL/Kinetic/Cartesian.h>

Types

Kinetic::Cartesian<FunctionKernel>:: Certificate

This is a model of *Kinetic::Certificate*.

Kinetic::Cartesian<FunctionKernel>:: Point_1

Kinetic::Cartesian<FunctionKernel>:: Point_2

Kinetic::Cartesian<FunctionKernel>:: Point_3

Kinetic::Cartesian<FunctionKernel>:: Weighted_point_3

The following are functors which generate *Certificate* objects. Each has a corresponding *_object* method which creates the functor. They are models of *Kinetic::CertificateGenerator*.

Kinetic::Cartesian<FunctionKernel>:: Orientation_2

Kinetic::Cartesian<FunctionKernel>:: Orientation_3

Kinetic::Cartesian<FunctionKernel>:: Side_of_oriented_circle_2

Kinetic::Cartesian<FunctionKernel>:: Side_of_oriented_sphere_3

Kinetic::Cartesian<FunctionKernel>:: Power_test_3

Kinetic::Cartesian<FunctionKernel>:: Weighted_orientation_3

Kinetic::Cartesian<FunctionKernel>:: Compare_x_1

Kinetic::Cartesian<FunctionKernel>:: Compare_x_2

Kinetic::Cartesian<FunctionKernel>:: Compare_y_2

Kinetic::Cartesian<FunctionKernel>:: Compare_x_3

Kinetic::Cartesian<FunctionKernel>:: Compare_y_3

Kinetic::Cartesian<FunctionKernel>:: Compare_z_3

Kinetic::Cartesian<FunctionKernel>:: Compare_distance_2

Kinetic::Cartesian<FunctionKernel>:: Compare_distance_3

Is Model for the Concepts

Kinetic::Kernel.

Concept

Kinetic::Certificate

Definition

The concept represents certificate. Its main purpose is to provide a way of creating *Time* objects corresponding to when the certificate fails and to cache any useful work done in find the *Time* for later.

Operations

Time	*c.failure_time()*	Returns the next failure time.
bool	*c.will_fail()*	Returns true if the certificate will ever fail.
void	*c.pop_failure_time()*	Advances to the next failure time (the next root of the certificate functions).

See Also

Kinetic::Kernel.

Kinetic::CertificateGenerator

Concept

Definition

This functor allows you to create certificate objects of some type. The models of this “concept” take some set of arguments which depend on the certificate being computed (for example three points for a two dimensional orientation) followed by either one or two instances of the *Kinetic::Simulator::Time* concept. The functions either return a *Certificate* or the corresponding value at the current time (if only a time value rather than an interval is passed).

Operations

Certificate *a(Args, Time begin, Time end)*

Return a *Certifate* object for the corresponding certificate.

CGAL::Sign *a(Args, Time t)* Compute the sign of the function at $\lim_{\delta \to 0} t + \delta$. This can be used to evaluate predicates at the current moment.

Has Models

All over the place.

See Also

Kinetic::KineticKernel

Example

Here you see how to use both functions on an orientation predicate.

```
KineticKernel::Point_2 a,b,c;
Simulator::Handle sh;
KineticKernek kk;

KineticKernel::Orientation_2 o2= kk.orientation_2_object();

KineticKernel::Certificate c= o2(a,b,c, sh->current_time(),
    sh->end_time());
if (c.will_fail()) {
  std::cout << "Certificate will fail" << std::endl;
}
// Compute the sign immediately following the current time
CGAL::Sign sn= o2(a,c,b, sh->current_time());
CGAL_postcondition(sn==CGAL::NEGATIVE);
```

Concept

Kinetic::FunctionKernel::ConstructFunction

Definition

The concept is used to construct functions.

Operations

Function	*a(NT a, ...)*	This family of methods takes a list of coefficients and returns a function. There can be any number of coeffients passed as arguments (up to about 25 in the current implementations).

See Also

FunctionKernel

Example

```
Function_kernel fk;
Function_kernel::Construct_function cf= fk.construct_function_object();
Function_kernel::Function f= cf(0,1,2,3,4,5);
```

Kinetic::EventQueue

Concept

Definition

The for priority queues used by the *Simulator*. The concept basically defines a priority queue which supports deletions and changes of items in the queue (but not their priorities). Items in the queue must implement the *Event* concept.

Types

Kinetic::EventQueue:: Key	The type used to access items in the queue in order to change or delete them.
Kinetic::EventQueue:: Priority	The priority type for items in the queue. This is typically the same as *Kinetic::Simulator::Time*

.

Creation

Kinetic::EventQueue q(Priority start, Priority end, int size_hint);

Construct a queue which will start at time start and run until time end.

Operations

template <class Event>
Key *q.insert(Priority, Event)*

Insert an event into the event queue. A *Key* which can be used to manipulated the event is returned.

void	*q.erase(Key)*	Erase an event from the queue.
template <class Event> *void*	*q.set(Key, Event)*	Change the data in the event referred to by the key.
template <class Event> *Event&*	*q.get(Key) const*	Access the event referred to by the passed key.
Priority	*q.priority(Key) const*	Return the priority of the event.
bool	*q.empty()*	Return true if the queue is empty.

Priority	*q.next_priority() const*	
		Return the priority of the next event in the queue.
void	*q.process_next()*	Process the next *Event* by calling its process method with its *Priority*.
void	*q.set_end_priority()*	Set the priority beyond which to ignore events.
bool	*q.contains(Key) const*	
		Return true if the queue contains the event and false if it does not. This is used for auditing events and can be slow if needed.

Has Models

Kinetic::Two_list_pointer_event_queue<FunctionKernel>, *Kinetic::Heap_pointer_event_queue< FunctionKernel>*.

Kinetic::Simulator::Event

Concept

Definition

The concept represents a single event. Models of should be passed to the *Kinetic::Simulator* when scheduling events which will in turn pass them to the *EventQueue*.

Operations

void	*a.process()*	This method is called when the event occurs. This method will only be called once per time this event is scheduled and the event will be removed from the queue immediately afterwards.
*void**	*a.kds()*	Return a *void* * which represents the KDS which this event belongs to. The pointer is used solely to tell if two events come from the same KDS for the purposes of handling degeneracy.
CGAL::Comparison_result	*a.compare_concurrent(Key a, Key b) const*	The two events *a* and *b* occur at the same time (*this* has key *a*). This method returns a *CGAL::Comparison_result* which is used to order the two equal events. If *CGAL::EQUAL* is returned then *merge* will be called.
bool	*a.merge_concurrent(Key a, Key b)*	The two events *a* and *b* occur at the same time (*this* has key *a*) and cannot be perturbed to be unequal. This event allows the KDS to merge event *b* with *a*. If it returns *true* then *b* is dropped from the event queue.
void	*a.audit(Key this_key)*	Audit that this is a valid event. To use this, kinetic data structure can check that this event is indeed pointed to by the correct part of the combinatorial structure.
std::ostream&	*a.write(std::ostream&) const*	Write the event to a stream.

Has Models

All over the place. *Kinetic::Event_base*.

See Also

Kinetic::EventQueue

Example

All of the kinetic data structures provided have models of Event. Here is the code implementing a swap event from the sorting kinetic data structure. Events occuring at equal times are perturbed so that the one that occurs first in the list is processed first (just to illustrate the idea).

```
template <class Certificate, class Id, class Root_enumerator>
class Swap_event {
  typedef Swap_event<class Certificate, class Id, class Root_enumerator>
   This;
public:
  Swap_event(Id o, Sort* sorter,
             const Certificate &s): left_object_(o),
                                    sorter_(sorter),
                                    s_(s){}
  void process(){
    sorter_->swap(left_object_, s_);
  }
  void *kds() const {return sorter_;}
  CGAL::Comparison_result perturb_comparison(typename Sort::Event_key a,
   typename Sort::Event_key b) const {
    return CGAL::compare(std::distance(sorter_->objects_begin(),
   left_object_),
                         std::distance(sorter_->objects_begin(),

   sorter_->simulator_handle()->get_event<This>(b).left_object_));
  }
  bool merge(typename Sort::Event_key a, typename Sort::Event_key b) {
    return false;
  }
  Id left_object_;
  Sort* sorter_;
  Certificate s_;
};
```

Kinetic::FunctionKernel

Definition

The concept Kinetic::FunctionKernel encapsulates all the methods for representing and handing functions. The set is kept deliberately small to easy use of new Kinetic::FunctionKernels, but together these operations are sufficient to allow the correct processing of events, handling of degeneracies, usage of static data structures, run-time error checking as well as run-time verification of the correctness of kinetic data structures. The computation of a polynomial with the variable negated is used for reversing time in kinetic data structures and can be omitted if that capability is not needed.

Types

Kinetic::FunctionKernel:: Function — The type of function being handled.

Kinetic::FunctionKernel:: NT — The basic representational number type.

Kinetic::FunctionKernel:: Root — A type representing the roots of a *Function*.

Kinetic::FunctionKernel:: Root_stack — A model of *RootStack*. These objects can be created by calling the *root_stack_object* method with a *Function* and two (optional) *Root* objects. The enumerator then enumerates all roots of the function in the open inverval defined by the two root arguments. They optional arguments default to positive and negative infinity.

Kinetic::FunctionKernel:: Root_enumerator_traits

The traits for the *Root_enumerator* class.

Each of the following types has a corresponding *type_object* method (not explicitly documented) which takes a *Function* as an argument.

Kinetic::FunctionKernel:: Sign_at — A functor which returns the sign of a *Function* at a *NT* or *Root*.

Kinetic::FunctionKernel:: Sign_after — A functor which returns sign of a function immediately after a root.

The following type is used to construct functions from a list of coefficients. To get an instance use the *construct_function_object()* method.

Kinetic::FunctionKernel:: Construct_function — This functor can be used create instances of *Function*. See its reference page *FunctionKernel::ConstructFunction* for more details.

The following functor likewise have a *type_object* method, but these take arguments other than a *Function*. The arguments are given below.

Kinetic::FunctionKernel:: Sign_between_roots — This functor, creation of which requires two *Root*s, returns the sign of a passed function between the pair of roots.

Kinetic::FunctionKernel:: Differentiate — This functor computes the derivitive of a *Function*. Construction takes no arguments.

The following methods do not require any arguments to get the functor and take one *Function* as a functor argument.

Kinetic::FunctionKernel:: Negate_variable — Map $f(x)$ to $f(-x)$.

Has Models

POLYNOMIAL::Kernel<RootStack>, *POLYNOMIAL::Filtered_kernel<RootStack>*.

See Also

Kinetic::RootEnumerator.

Example

We provide several models of the concept, which are not documented separately. The models of *Kinetic::SimulationTraits* all choose appropriate models. However, if more control is desired, we here provide examples of how to create the various supported *Kinetic::FunctionKernel*.

A Sturm sequence based kernel which supports exact comparisons of roots of polynomials (certificate failure times):

```
typedef CGAL::POLYNOMIAL::Polynomial<CGAL::Gmpq> Function;
typedef CGAL::POLYNOMIAL::Sturm_root_stack_traits<Function>
   Root_stack_traits;
typedef CGAL::POLYNOMIAL::Sturm_root_stack<Root_stack_traits> Root_stack;
typedef CGAL::POLYNOMIAL::Kernel<Function, Root_stack> Function_kernel;
```

A wrapper for *CORE::Expr* which implements the necessary operations:

```
typedef CGAL::POLYNOMIAL::CORE_kernel Function_kernel;
```

A function kernel which computes approximations to the roots of the polynomials:

```
typedef CGAL::POLYNOMIAL::Polynomial<double> Function;
typedef CGAL::POLYNOMIAL::Root_stack_default_traits<Function>
   Root_stack_traits;
typedef CGAL::POLYNOMIAL::Numeric_root_stack<Root_stack_traits>
   Root_stack;
typedef CGAL::POLYNOMIAL::Kernel<Function, Root_stack> Function_kernel;
```

When using the function kernel in kinetic data structures, especially one that is in exact, it is useful to wrap the root stack. The wrapper checks the sign of the certificate function being solved and uses that to handle degeneracies. This is done by, for the inexact solvers

```
typedef Kinetic::Derivitive_filter_function_kernel<Function_kernel>
   KDS_function_kernel;
```

and for exact solvers

```
typedef Kinetic::Handle_degeneracy_function_kernel<Function_kernel>
   KDS_function_kernel;
```

For exact computations, the primary representation for roots is the now standard choice of a polynomial with an associated isolating interval (and interval containing exactly one distinct root of a polynomial) along with whether the root has odd or even multiplicity and, if needed, the Sturm sequence of the polynomial. Two intervals can be compared by first seeing if the isolating intervals are disjoint. If they are, then we know the ordering of the respective roots. If not we can subdivide each of the intervals (using the endpoints of the other interval) and repeat. In order to avoid subdividing endlessly when comparing equal roots, once we subdivide a constant number of times, we use the Sturm sequence of p and $p'q$ (where p and q are the two polynomials and p' is the derivative of p) to evaluate the sign of the second at the root of the first one directly (note that this Sturm sequence is applied to a common isolating interval of the roots of interest of both polynomials).

Concept

Kinetic::FunctionKernel::Function

Definition

The concept represents a function.

Types

Function:: NT	The number type used in describing the function.
Function a(NT);	Construct a constant function from a number.

Operations

NT	*a(NT)*	Evaluate the function at an *NT*.

See Also

FunctionKernel, *FunctionKernel::ConstructFunction*

Example

Several ways to create functions:

Using *Kinetic::ConstructFunction*:

```
Traits::Function_kernel::Construct_function cf=
   traits.function_kernel_object().construct_function_object();
Traits::Kinetic_kernel::Motion_function x= cf(0.0,1.0,2.0);
Traits::Kinetic_kernel::Motion_function y= cf(0.0,1.0,2.0);
Traits::Kinetic_kernel::Point_2 pt(x,y);
```

Using the constructor:

```
double coefs[]={1.0, 2.0, 3.0};
Traits::Kinetic_kernel::Motion_function z(coefs, coefs+3);
```

Using ring operations:

```
Traits::Kinetic_kernel::Motion_function z= x*z+y;
```

CGAL::Kinetic::Default_instantaneous_kernel<SimulationTraits>

Class

Definition

This class provides a model of the *Kinetic::InstantaneousKernel* for use with general Cartesian Geometry. It provides all the predicates needed for Delaunay triangulations and regular triangulations.

#include <CGAL/Kinetic/Default_instantaneous_kernel.h>

Types

Kinetic::Default_instantaneous_kernel<SimulationTraits>:: Orientation_2

Kinetic::Default_instantaneous_kernel<SimulationTraits>:: Orientation_3

Kinetic::Default_instantaneous_kernel<SimulationTraits>:: Side_of_oriented_circle_2

Kinetic::Default_instantaneous_kernel<SimulationTraits>:: Side_of_oriented_sphere_3

Kinetic::Default_instantaneous_kernel<SimulationTraits>:: Power_test_3

Kinetic::Default_instantaneous_kernel<SimulationTraits>:: Weighted_orientation_3

Kinetic::Default_instantaneous_kernel<SimulationTraits>:: Compare_x_1

Kinetic::Default_instantaneous_kernel<SimulationTraits>:: Compare_x_2

Kinetic::Default_instantaneous_kernel<SimulationTraits>:: Compare_y_2

Kinetic::Default_instantaneous_kernel<SimulationTraits>:: Compare_x_3

Kinetic::Default_instantaneous_kernel<SimulationTraits>:: Compare_y_3

Kinetic::Default_instantaneous_kernel<SimulationTraits>:: Compare_z_3

Kinetic::Default_instantaneous_kernel<SimulationTraits>:: Compare_distance_2

Kinetic::Default_instantaneous_kernel<SimulationTraits>:: Compare_distance_3

Kinetic::Default_instantaneous_kernel<SimulationTraits>:: Coplanar_orientation_3

Note that this one does not work if the current time is not a *NT*.

Kinetic::Default_instantaneous_kernel<SimulationTraits>:: Coplanar_side_of_bounded_circle_3

Note that this one does not work if the current time is not a *NT*.

Is Model for the Concepts

Kinetic::InstantaneousKernel

Kinetic::InstantaneousKernel

Definition

The concept Kinetic::InstantaneousKernel covers models that act as adaptors allowing CGAL static data structures to act on snapshots of kinetic data. Different methods for evaluating predicates are used depending on whether time is set using an *NT* or a *Time* object. Evaluating predicates when time is the former is much cheaper.

Types

Kinetic::InstantaneousKernel:: NT	A number type which can be used to represent the current time. This must be a ring or field type.
Kinetic::InstantaneousKernel:: Time	The type used to represent the current time. This type must be comparable.

Operations

Time	*a.time()*	Return the current time.
NT	*a.time_as_nt()*	Return the current time as an *NT*. As a precondition, *time_is_nt* must be true.
bool	*a.time_is_nt()*	Return true if the last time time was set, it was using an object of type *NT*.
void	*a.set_time(Time)*	Set the current time to have a certain value. All existing predicates are updated automatically.
void	*a.set_time(NT)*	The the current time to be an instance of *NT*. With this more efficient techniques can be used. *time_is_nt()* must be true.
Static_object	*a.static_object(Key)*	Return a static object corresponding to the kinetic object at this instant in time. *time_is_nt()* must be true.

Has Models

Kinetic::Default_instantaneous_kernel

Key

Definition

The concept Key is a unique identifier for something in some sort of table. In general, they can be only created by the table and are returned when a appropriate *new_foo()* method is called on the table. There are two classes of values for a Key, valid and invalid. The latter cannot refer to something in a table. Use the method *is_valid()* to differentiate.

Key a;	The default constructor is guaranteed to construct an invalid key (i.e. one which is false when cast to a bool.

Operations

bool	*a.is_valid() const*	This method returns false if the key was created using the default constructor or was otherwise created to be invalid.
std::ostream&	*std::ostream& << Event*	

Write a text description of the key to a standard stream.

Has Models

Kinetic::Simulator::Event_key, *Kinetic::Active_objects_vector<Object>::Key*.

Kinetic::Kernel

Concept

Definition

The concept Kinetic::Kernel acts as the kinetic analog of a CGAL kernel. It provides some set of primitives and predicats acting on them. The predicates are instances of *Kinetic::CertificateGenerator* and can be used to either create *Certificate*s or to evaluate instantaneous predicates.

Types

Kinetic::Kernel:: Motion_function	The type which is used to represent coordinates of moving primitives. It is a model of the concept *FunctionKernel::Function*. This is the analog of the CGAL kernel *RT*.
Kinetic::Kernel:: Certificate	The type representing the results of predicates. See *Kinetic::Certificate*.
Kinetic::Kernel:: Function_kernel	The type of the function kernel used. See *Kinetic::FunctionKernel*.

Operations

Function_kernel *kk.function_kernel_object() const*

Gets a copy of the function kernel.

Has Models

Kinetic::Cartesian<FunctionKernel>.

Class

CGAL::Listener<Interface>

Definition

The *Listener<Interface>* class provides the core of the run time notification system used by the kinetic data structures package. In short, notifications are handled through proxy objects called listeners. In order to listen for notifications from an object, called the notifier, you make define a small class called a listener proxy, which inherits from the Listener interface defined by the notifier. When constructing your listner poxy, you pass a reference counted pointer to the notifier, which is used to register the proxy for notifications. When a notification occurs, the notifier calls the *new_notification* method on the proxy, passing the type of the notification. The proxy stores a reference counted pointer to the notifier, ensuring that there are never any dangling pointers in the system.

The class *Listener<Interface>* provides base class for listener proxy objects. A notifier should provide a class which inherits from this base. To use this base class, implement a class, here called *Interface*, which defines a type *Interface::Notification_type* and a type *Interface::Notifier_handle*.

The *Notification_type* is generally an enum with one value for each type of notification which can be used.

The *Notifier_handle* is the type of a (ref counted) pointer to the object providing the notifications. The ref counter pointer must provide a nested type *Pointer* which is the type of a raw pointer.

The *Listener<Interface>* maintains a ref counted pointer to the object performing notifications. It is registered for notifications on construction and unregistered on destruction using the function *set_listener(Listener< Interface>*)* on the object providing the notifications. The use of ref counted pointers means that as long as the notification object exists, the object providing the notifications must exist, ensuring that the object providing the notifications is not prematurely destroyed.

These objects cannot be copied since the notifier only support one listener. If copying and more than one listener are desired, the *Multi_listener<Interface>* base class should be used instead.

As a side note, Boost provides a similar functionality in the Boost.Signal package. However, it is quite a bit more complex (and flexible). This complexity add significantly to compile time and (although I did not test this directly), I suspect it is much slower at runtime due to the overhead of worrying about signal orders and not supporting single signals. In addition, it does not get on well with Qt due to collisions with the Qt moc keywords.

There is also the TinyTL library which implements signals. As of writing it did not have any easy support for making sure all pointers are valid, so it did not seem to offer significant code saving over writing my own.

#include <CGAL/Kinetic/Listener.h>

Types

Listener<Interface>:: Notifier_handle	This type is inherited from the *Interface* template argument. It is a reference counted pointer type for the object providing notifications.
Listener<Interface>:: Notification_type	The type (usually an enum) used to distinguish different types of notifications. This is inherited from the *Interface* template argument.

Creation

Listener<Interface> l(Notifier_handle np);

The *Listener<Interface>* subscribes to events coming from the notifier and stores a pointer to the notifier.

Operations

Notifier_handle *l.notifier()*

Return a pointer to the notifier.

virtual void *l.new_notification(Notification_type)*

This method is pure virtual. A class which wishes to receive events must inherit from this class and implement this method. The method will then be called whenever there is a notification.

See Also

Multi_listener<Interface>.

Example

Here is a simplier class that provides notifications:

```
struct Notifier: public CGAL::Kinetic::Ref_counted<Notifier>
{
public:
  Notifier(): data_(0), listener_(NULL){}

  struct Listener_interface
  {
  public:
    typedef enum Notification_type {DATA_CHANGED} Notification_type;
    typedef Notifier::Handle Notifier_handle;
  };

  typedef CGAL::Kinetic::Listener<Listener_interface> Listener;
  friend class CGAL::Kinetic::Listener<Listener_interface>;

  void set_data(int d) {
    data_=d;
    if (listener_ != NULL)
   listener_->new_notification(Listener_interface::DATA_CHANGED);
  }

protected:
  void set_listener(Listener *l) {
    listener_= l;
  }
  Listener* listener() const {return listener_;}
```

```
  int data_;
  Listener *listener_;
};
```

Now the listener:

```
struct My_listener: public Notifier::Listener, public
    CGAL::Kinetic::Ref_counted<My_listener>
{
  typedef Notifier::Listener::Notifier_handle PP;
  My_listener(PP p): P(p){}

  void new_notification(P::Notification_type nt) {
    ...
  }
};
```

CGAL::Multi_listener<Interface>

Class

Definition

The class *Multi_listener<Interface>* implements a base class for listeners where more than one listener is allowed to subscribe to a notifier. See *Listener* for full documentation. This uses the function calls *new_listener()* and *delete_listener()* to register and unrester the listener (instead of *set_listener()*).

#include <CGAL/Kinetic/Multi_listener.h>

See Also

Listener<Interface>.

Class

CGAL::Ref_counted<T>

Definition

The class *Ref_counted<T>* implements a base class for objects which are reference counted. To use it simply inherit from *Ref_counted<T>* (passing the type to be reference counted as the template argument) and then access the object through *Handle* objects rather than bare C++ pointers.

#include <CGAL/Kinetic/Ref_counted.h>

Types

Ref_counted<T>:: Handle	A reference counted pointer to an Object.
Ref_counted<T>:: Const_handle	A const reference counted pointer to an Object.

Creation

Ref_counted<T> rc;	default constructor.

Operations

There are no methods which should be called by users of this class.

Kinetic::RootStack

Concept

Definition

The concept Kinetic::RootStack enumerates through roots of a function contained in a half open interval [lb. . . ub).

Types

Kinetic::RootStack:: Root	The root of a function.
Kinetic::RootStack:: Traits	The traits class for this concept.

Creation

Kinetic::RootStack re; — default constructor.

Kinetic::RootStack re(Function f, Root lb, Root ub, Traits tr);

Construct a Kinetic::RootStack over the roots of *f* in the half open interval [*lb* to *ub*).

Operations

void	*re.pop()*	Advance to the next root. As a precondition, empty() must be false.
Root	*re.top()*	Return the current root. As a precondition, empty() must be false. Note that the *Root* returned might not actually be in the interval (since the solver has not yet proved that there are no more roots).
bool	*re.empty()*	Return true if there are known to be no more roots left. There might not actually be any roots of the polynomial left in the interval, but the work necessary to prove this has been delayed.

See Also

Kinetic::FunctionKernel, *Kinetic::Certificate*.

Concept

Kinetic::SimulationTraits

Definition

This concept ties together the parts needed in order to run a kinetic data structure. We provide several models of this concept:

- *Kinetic::Exact_simulation_traits*
- *Kinetic::Inexact_simulation_traits*
- *Kinetic::Regular_triangulation_exact_simulation_traits*
- *Kinetic::Regular_triangulation_inexact_simulation_traits*

All support trajectories defined by polynomial coordinates. The *Exact* vs *Inexect* picks whether the roots of the certificate functions are compared exactly or approximated numerically. The regular triangulation models have weighted points of the appropriate dimension as the primitive used in the *Kinetic::InstantaneousKernel* and the *Kinetic::ActiveObjectsTable*.

Types

Kinetic::SimulationTraits:: NT — The number type used for representation.

Kinetic::SimulationTraits:: Instantaneous_kernel

A model of *Kinetic::InstantaneousKernel* which can be used to apply static CGAL data structures to snapshots of moving data.

Kinetic::SimulationTraits:: Kinetic_kernel — A model of *Kinetic::Kernel*.

Kinetic::SimulationTraits:: Function_kernel — A model of *Kinetic::FunctionKernel*.

Kinetic::SimulationTraits:: Active_points_[123]_table

A model of *Kinetic::ActiveObjectsTable* which holds the relevant kinetic primitives.

Kinetic::SimulationTraits:: Simulator — A model of *Kinetic::Simulator* which will be used by all the kinetic data structures.

Operations

Instantaneous_kernel *st.instantaneous_kernel_object()*

Get a new instantaneous kernel.

Kinetic_kernel *st.kinetic_kernel_object()*

Get a new kinetic kernel.

Function_kernel *st.function_kernel_object()*

Get a new function kernel.

Simulator::Handle *st.simulator_handle()* Return a pointer to the *Kinetic::Simulator* which is to be used in the simulation.

Active_points_[123]_table::Handle

st.active_points_[123]_table_handle()

Return a pointer to the *Kinetic::ActiveObjectsTable* which is to be used in the simulation.

Has Models

Kinetic::Exact_simulation_traits, *Kinetic::Inexact_simulation_traits*, *Kinetic::Regular_triangulation_exact_simulation_traits*, *Kinetic::Regular_triangulation_inexact_simulation_traits*

Example

The simulation traits class is simply there for convenience in order to bundle a set of related typedefs and create a few objects. As a resulting, creating your own requires little though, and just copying and changing a few lines. An example is below which sets up to use the CORE Sturm sequences to solve polynomials rather than our own (faster) solvers. It can be found in examples/Kinetic_framework/defining_a_simulation_traits.cpp.

```
#include <CGAL/Polynomial/Sturm_root_stack_traits.h>
#include <CGAL/Polynomial/Sturm_root_stack.h>
#include <CGAL/Kinetic/Active_objects_vector.h>
#include <CGAL/Kinetic/Default_instantaneous_kernel.h>
#include <CGAL/Kinetic/Cartesian.h>
#include <CGAL/Kinetic/Handle_degeneracy_function_kernel.h>
#include <CGAL/Kinetic/Default_simulator.h>
#include <CGAL/Kinetic/Two_list_pointer_event_queue.h>
#include <CGAL/Exact_predicates_exact_constructions_kernel.h>

using namespace CGAL::Kinetic;

struct My_simulation_traits {
  typedef My_simulation_traits This;

  typedef CGAL::Exact_predicates_exact_constructions_kernel
   Static_kernel;
  //typedef
   CGAL::Regular_triangulation_euclidean_traits_3<Static_kernel_base>
   Static_kernel;
  typedef CGAL::POLYNOMIAL::Polynomial<Static_kernel::FT> Function;
  typedef CGAL::POLYNOMIAL::Sturm_root_stack_traits<Function>
   Root_stack_traits;
  typedef CGAL::POLYNOMIAL::Sturm_root_stack<Root_stack_traits>
   Root_stack;
  typedef CGAL::POLYNOMIAL::Kernel<Function, Root_stack> Function_kernel;

  typedef
   CGAL::Kinetic::Handle_degeneracy_function_kernel<Function_kernel,
   false>  Simulator_function_kernel_base;
  struct Simulator_function_kernel: public
   Simulator_function_kernel_base{};

  typedef Cartesian<Simulator_function_kernel> Kinetic_kernel;
  typedef Two_list_pointer_event_queue<Function_kernel> Event_queue;
  typedef Default_simulator<Simulator_function_kernel, Event_queue >
   Simulator;

  typedef Active_objects_vector<Kinetic_kernel::Point_1>
   Active_points_1_table;
  typedef Active_objects_vector<Kinetic_kernel::Point_2>
   Active_points_2_table;
  typedef Active_objects_vector<Kinetic_kernel::Point_3>
   Active_points_3_table;
  // typedef Active_objects_vector<Kinetic_kernel::Weighted_point_3>
   Active_weighted_points_3_table;
```

```
  typedef Default_instantaneous_kernel<This> Instantaneous_kernel;

  Active_points_1_table* active_points_1_table_handle() const { return
    ap1_.get();}
  Active_points_2_table* active_points_2_table_handle() const {return
    ap2_.get();}
  Active_points_3_table* active_points_3_table_handle() const {return
    ap3_.get();}
  //Active_weighted_points_3_table*
    active_weighted_points_3_table_handle() const {return awp3_.get();}

  Simulator* simulator_handle() const { return sim_.get();}
  const Static_kernel& static_kernel_object() const {return k_;}
  const Kinetic_kernel& kinetic_kernel_object() const {return kk_;}

  Instantaneous_kernel instantaneous_kernel_object() const {
    return Instantaneous_kernel(*this);
  }

  My_simulation_traits(const Simulator::Time &lb,
                       const Simulator::Time &ub): sim_(new
    Simulator(lb, ub)),
                                                   ap1_(new
    Active_points_1_table()),
                                                   ap2_(new
    Active_points_2_table()),
                                                   ap3_(new
    Active_points_3_table())
                                                   //awp3_(new
    Active_weighted_points_3_table())
{}


  bool is_exact() const {
    return true;
  }
protected:
  Simulator::Handle sim_;
  Active_points_1_table::Handle ap1_;
  Active_points_2_table::Handle ap2_;
  Active_points_3_table::Handle ap3_;
  //Active_weighted_points_3_table::Handle awp3_;
  Static_kernel k_;
  Kinetic_kernel kk_;
  Function_kernel fk_;
};

#endif
```

Class

CGAL::Kinetic::Default_simulator<FunctionKernel, EventQueue>

Definition

The class *Kinetic::Default_simulator<FunctionKernel, EventQueue>* controls kinetic data structures by maintaining a concept of time and ensuring that events are processed when necessary.

#include <CGAL/Kinetic/Default_simulator.h>

Is Model for the Concepts

Kinetic::Simulator.

Creation

Kinetic::Default_simulator<FunctionKernel, EventQueue> sim(const Time start=Time(0), const Time end= Time::infinity())

Construct a *Kinetic::Default_simulator<FunctionKernel, EventQueue>* which will process events between times start and end (events outside this window will be discarded).

CGAL::Kinetic::Simulator_listener<Listener, KDS>

Class

Definition

The class *Kinetic::Simulator_listener<Listener, KDS>* acts as a helper class for kinetic data structures which want to respond to *Simulator::Listener::HAS_AUDIT_TIME* notifications. When kinetic data structures can audit themselves, the *Kinetic::Simulator_listener<Listener, KDS>* calls the *audit()* method on the kinetic data structure.

#include <CGAL/Kinetic/listeners.h>

Creation

*Kinetic::Simulator_listener<Listener, KDS> a(Simulator::Handle, KDS *kds);*

default constructor.

See Also

Kinetic::Simulator, *Listener<Interface>*.

CGAL::Kinetic::Simulator_objects_listener<Simulator_listener, KDS>

Definition

The class *Kinetic::Simulator_objects_listener<Simulator_listener, KDS>* is a helper for classes which wish to react to *Simulator::Listener::DIRECTION_OF_TIME* notifications. The helper object translates such notifications *reverse_time* function calls on the responder. See *Kinetic::Qt_moving_points_2* for a simple example of using this helper function.

#include <CGAL/Kinetic/Simulator_objects_listener.h>

Creation

Kinetic::Simulator_objects_listener<Simulator_listener, KDS> a(Simulator::Handle, KDS);*

default constructor.

See Also

Kinetic::Listener.

Kinetic::Simulator

Definition

The class Kinetic::Simulator controls kinetic data structures by maintaining a the current time and ensuring that events are processed when necessary.

In addition, the Kinetic::Simulator can call on the kinetic data structures to audit themselves at appropriate times. When the last event processed and the next to be processed have different times, then there is a rational value of time at which all kinetic data structures should be non-degenerate (since there are no events at that time). At such a time, kinetic data structures can easily verify their correctness by checking that all the certificate predicates have the correct value. When exactness checks are enabled, whenever the last event processed and the next event to be processed have different times, a *Kinetic::Simulator::Listener::HAS_AUDIT_TIME* notification is made. Kinetic data structures can listen for that event, and when it is made, they can call *Kinetic::Simulator::audit_time()* to get the time value and then verify that their structure is correct.

In addition, at such a time, the *Event::audit(Key)* is called on each event. This allows kinetic data structures to check that the event should be in the queue.

Typically, the simulator is created by the Kinetic:SimulationTraits class and kinetic data structures request a handle to it from there.

Events that occur at or after *Kinetic::Simulator::end_time()* will may not be processed. The exception are events which are scheduled using the *new_final_event(Event)* call which are guaranteed to occur after all other events (but have no particular order amongst themselves).

Types

Kinetic::Simulator:: Function_kernel	The type of the function kernel used to instantiate this Kinetic::Simulator.
Kinetic::Simulator:: Listener	Extend this base class to listen to notifications from this Kinetic::Simulator. There are two types of notifications: *HAS_AUDIT_TIME* and *DIRECTION_OF_TIME*. The first is made when kinetic data structures can perform an audit. The second is made when the direction of time is changed.
Kinetic::Simulator:: Time	The representation type for times in the simulator. It is *Function_kernel::Root_enumerator::Root*.
Kinetic::Simulator:: Event_key	The key to access scheduled *Event* in order to inspect or delete them.
Kinetic::Simulator:: NT	The basic number type used in computations.
Kinetic::Simulator:: Handle	A reference counted pointer to be used for storing references to the object.
Kinetic::Simulator:: Const_handle	A reference counted pointer to be used for storing references to the object.

Creation

Kinetic::Simulator sim(const Time start=Time(0), const Time end= Time::infinity());

Construct a Kinetic::Simulator which will process events between times start and end (events outside this window will be discarded).

Operations

Function_kernel *sim.function_kernel_object() const*

Access the *Function_kernel* object used by the Kinetic::Simulator.

Time *sim.current_time()* Return the current time.

void *sim.set_current_time(Time t)*

Set the current time to *t*, which cannot be less than *current_time*. Any events in the queue before time *t* are processed.

NT *sim.next_time_reprsentable_as_nt() const*

This function returns a time which can be represented using an instance of type *NT* which is shortly after the current time. You can then advance the current time to this one and act on the data structure using the return nt.

bool *sim.has_current_time_as_nt() const*

Return true if there is a rational number which is equivalent to the current time. Equivalent means that it has the same ordering relation to all previous and scheduled events.

bool *sim.has_audit_time() const*

Returns true if the current time is a rational number and there are no events at the current time. This means that the simulation can be audited at this time.

Time *sim.next_event_time() const*

Return the time of the next event in the queue.

Time *sim.end_time() const* Return the time the simulation will end. If time is running backwards, then this returns *Time::infinity()*.

void *sim.set_interval(Time t_cur, Time t_end)*

Set the current time to t_{cur} and the end time to t_{end}. The event queue must be empty. Use this method if you want to reset or extend the simulation.

template <class Event>
Event_key *sim.new_event(Time t, const Event event)*

Schedule a new event at time *t*. The object *event* must implement the concept *Event*. The *Event_key* returned can be used to access or deschedule the event.

template <class Event>
Event_key *sim.new_final_event(const Event event)*

Schedule a new event that will occur at the end of the simulation. This type of event is useful if, for example, you want to run for a while, change all motions, and then run some more.

Event_key *sim.null_event() const* This method returns an *Event_key* which is guaranteed never to be assigned to any real event. This is a very useful placeholder for events which are known never to occur (and allows data structures to differentiate between uninitialized and never failing).

void *sim.audit_event(Event_key) const*

Performs some checks as to whether the key corresponds to a valid event. Generally, a event is valid if it is not defaultly constructed and either is in the queue or is the *null_event()*.

bool *sim.empty() const* Return true if there are no more events.

void *sim.delete_event(const Event_key k)*

Remove the event referenced by *k* from the event queue.

template <class Ev>
typename Queue::Event_handle<Ev>::Handle

sim.event(const Event_key k, const Ev e) const

This method returns a pointer to an event, which can be used for recoving data, such as cached solvers, from that event. The second argument really shouldn't be there, but gcc seems to sometimes have issues if you try to specify the template value directly.

Time	*sim.event_time(Event_key k) const*	
		Return the time at which the event referenced by *k* occurs.
template <class Ev> *Event_key*	*sim.set_event(Event_key k, const Ev ev)*	
		Set the event referenced by key *k* to *ev*, for example if you want to change what happens when that event occurs. A new event key is returned.
Sign	*sim.direction_of_time() const*	
		Return *POSITIVE* if time is running forwards or *NEGATIVE* if it is running backwards.
void	*sim.set_direction_of_time(Sign dir) const*	
		Set which direction time is running.
unsigned int	*sim.current_event_number() const*	
		Return the number of events which have been processed.
void	*sim.set_current_event_number(unsigned int i) const*	
		Process all events up to the ith event. *i* cannot be less than *current_event_number*.

See Also

Kinetic::Simulator_objects_listener<Simulator_listener, KDS>, *Kinetic::Simulator_kds_listener<Simulator_listener, KDS>*.

Has Models

Kinetic::Default_simulator<FunctionKernel, EventQueue>

Kinetic::Simulator::Time

Concept

Definition

The concept represents time in the simulator.

Time a(NT); Construct an instance of time from a number type, where NT is the number type used in the simulation.

Operations

std::ostream& *std::ostream& << Time*

Write it to a stream.

double *to_double(Time)*

Return a double approximation of the time value.

std::pair<double, double>

to_interval(Time)

Return an interval containing the time value.

Comparisons with other *Kinetic::Simulator::Time* objects are supported.

Has Models

double, *Kinetic::FunctionKernel::Root*

See Also

Kinetic::Simulator

Part XVII

Support Library

Chapter 73

STL Extensions for CGAL

Michael Hoffmann, Lutz Kettner, Sylvain Pion, and Ron Wein

Contents

CGAL is designed in the spirit of the generic programming paradigm to work together with the Standard Template Library (STL) [C++98, Aus98]. This chapter documents non-geometric STL-like components that are not provided in the STL standard but in CGAL: a doubly-connected list managing items in place (where inserted items are not copied), a compact container, a multi-set class that uses three-valued comparisons and offers additional functionality, generic algorithms, iterators, functor adaptors for binding and swapping arguments and for composition, functors for projection and creation and adaptor classes around iterators and circulators. See also circulators in Chapter 76. A class storing polymorphic objects is also provided, as well as a class to manage the uncertainty of some values. Finally, tags and policy classes to specify complexity trade-offs of data-structures, and a class which helps specifying that the default types in template parameter lists are desired is also provided.

73.1 Doubly-Connected List Managing Items in Place

The class *In_place_list<T,bool>* manages a sequence of items in place in a doubly-connected list. Its goals are the flexible handling of memory management and performance optimization. The item type has to provide the two necessary pointers `&`*T::next_link* and `&`*T::prev_link*. One possibility to obtain these pointers is to inherit them from the base class *In_place_list_base<T>*.

The class *In_place_list<T,bool>* is a container quite similar to STL containers, with the advantage that it is able to handle the stored elements by reference instead of copying them. It is possible to delete an element only knowing its address and no iterator to it. This used to simplify mutually pointed data structures like a halfedge data structure for planar maps or polyhedral surfaces (the current design does not need this anymore). The usual iterators are also available.

73.2 Compact Container

The class *Compact_container<T, Allocator>* is an STL like container which provides a very compact storage for its elements. It achieves this goal by requiring *T* to provide access to a pointer in it, which is going to be used by *Compact_container<T, Allocator>* for its internal management. The traits class *Compact_container_traits< T>* specifies the way to access that pointer. The class *Compact_container_base* can be used as a base class to provide the pointer, although in this case you do not get the most compact representation. The values that this pointer can have during valid use of the object are valid pointer values to 4 bytes aligned objects (i.e., the two least significant bits of the pointer need to be zero when the object is constructed). Another interesting property of this container is that iterators are not invalidated during *insert* or *erase* operations.

The main deviation from the STL container concept is that the ++ and -- operators of the iterator do not have a constant time complexity in all cases. The actual complexity is related to the maximum size that the container has had during its life time compared to its current size, because the iterator has to go over the "erased" elements as well, so the bad case is when the container used to contain lots of elements, but now has far less. In this case, we suggest to do a copy of the container in order to "defragment" the internal representation.

This container has been developed in order to efficiently handle large data structures like the triangulation and halfedge data structures. It can probably be useful for other kinds of graphs as well.

73.3 Multiset with Extended Functionality

The class *Multiset<Type,Compare,Allocator>* represents a multi-set of elements of type *Type*, represented as a red-black tree (see [CLRS01, Chapter 13] for an excellent introduction to red-black trees). It differs from the STL's *multiset* class-template mainly due to the fact that it is parameterized by a comparison functor *Compare* that returns the three-valued *Comparison_result* (namely it returns either *SMALLER*, *EQUAL*, or *LARGER*), rather than a *less* functor returning *bool*. Thus, it is possible to maintain the underlying red-black tree with less invocations of the comparison functor, which can considerably decrease running times, especially when comparing elements of type *Type* is an expensive operation.

Multiset<Type,Compare,Allocator> also guarantees that the order of elements sent to the comparison functor is fixed. For example, if we insert a new element *x* into the set (or erase an element from the set), then we always invoke *Compare() (x, y)* (and never *Compare() (y, x)*), where *y* is an element already stored in the set. This behavior, not supported by *std::multiset*, is sometimes crucial for designing more efficient comparison predicates.

The interface of *Multiset<Type,Compare,Allocator>* is in general derived from *std::multiset*. However, it extends the interface by offering some additional operations, such as: inserting of an element into the set given its *exact* position (and not just using an insertion hint); looking up keys whose type may differ from *Type*, as long as users supply a comparison functor *CompareKey*, between the keys and set elements; and catenating and splitting sets.

73.4 Polymorphic Object

The class *Object* can store an object of whatever other type. It can be used by a function to return objects of different types. A mechanism to extract the stored object based on its type is also provided. This class is similar to *boost::any*.

73.5 Uncertainty Management

The class *Uncertain*<*T*> represents a range of values of type *T*. *T* is allowed to stand for *bool*, or CGAL's enumeration types *Sign*, *Comparison_result*, *Orientation*, *Oriented_side*, *Bounded_side* and *Angle*.

The idea is that sometimes you are not sure of the result of a function, and you would like to communicate that to the caller. *Uncertain*<*T*> allows just that. It also provides functions to naturally extend the Boolean operations for *Uncertain*<*bool*> for example.

Uncertain<*T*> is used in CGAL as the return type of geometric predicates when the number type used is interval arithmetic like *Interval_nt*. End users typically do not see it as it is hidden in the implementation of the filtered predicates provided by the various filtered kernels, but it is important that providers of predicates that are meant to be filtered by *Filtered_predicate*, know about it.

It can also be used in other contexts as well, as it is a general tool.

73.6 Complexity tags and policies

Some data structures and algorithms can be implemented with different complexity trade-offs between memory usage and time complexity. CGAL provides the tags *Fast* and *Compact* which can be used to select between those variants. For example, the *Location_policy* class is parameterized by these tags and allows to specify the complexity of point location (currently in *Delaunay_triangulation_3* only). Convenient typedefs *Fast_location* and *Compact_location* are also provided.

73.7 Default Arguments in Template Parameter Lists

In C++, it is possible to specify defaults at the end of a template parameter list. Specifying that one wishes to use the default is simply done by omitting it. This is however possible only at the end of the list. *CGAL::Default* provides a simple mechanism that performs something equivalent anywhere in the sequence.

73.8 C++ 11 wrappers

Wrappers for the classes *array* and *tuple* which, based on availability, either use the version of *Boost* or the one provided by the standard library are provided in the namespace *CGAL::cpp11*. The namespace alias *CGAL::cpp0x* is provided for backward compatibility. Those are documented for completeness and implementers. They are not intended to be used by users of the library.

STL Extensions for CGAL Reference Manual

Michael Hoffmann, Lutz Kettner, Sylvain Pion, and Ron Wein

73.9 Classified Reference Pages

Doubly-Connected List Managing Items in Place

Compact Container

Multiset with Extended Functionality

Generic Algorithms

Iterators and Iterator/Circulator Adaptors.

Projection Function Objects

Creator Function Objects

Utilities

73.10 Alphabetical List of Reference Pages

Class

CGAL::In_place_list_base<T>

Definition

The node base classes provides pointers to build linked lists. The class *In_place_sl_list_base<T>* provides a pointer *next_link* for a single linked list. The class *In_place_list_base<T>* provides an additional pointer *prev_link* for doubly linked lists. These names conform to the default parameters used in the template argument lists of the container classes. The pointers are public members.

#include <CGAL/In_place_list.h>

Variables

*T**	*next_link;*	forward pointer
*T**	*prev_link;*	backward pointer

Class

CGAL::In_place_list<T,bool>

Definition

An object of the class *In_place_list<T,bool>* represents a sequence of items of type *T* that supports bidirectional iterators and allows constant time insert and erase operations anywhere within the sequence. The functionality is similar to the *list<T>* in the STL.

The *In_place_list<T,bool>* manages the items in place, i.e., inserted items are not copied. Two pointers of type *T** are expected to be reserved in *T* for the list management. The base class *In_place_list_base<T>* can be used to obtain such pointers.

The *In_place_list<T,bool>* does not copy element items during insertion (unless otherwise stated for a function). On removal of an item or destruction of the list the items are not deleted by default. The second template parameter *bool* is set to *false* in this case. If the *In_place_list<T,bool>* should take the responsibility for the stored objects the *bool* parameter could be set to *true*, in which case the list will delete removed items and will delete all remaining items on destruction. In any case, the *destroy()* member function deletes all items. Note that these two possible versions of *In_place_list<T,bool>* are not assignable to each other to avoid confusions between the different storage responsibilities.

#include <CGAL/In_place_list.h>

Parameters

The full class name is *In_place_list<T, bool managed = false, class Alloc = CGAL_ALLOCATOR(T)>*.

The parameter *T* is supposed to have a default constructor, a copy constructor and an assignment operator. The copy constructor and the assignment may copy the pointers in *T* for the list management, but they do not have to. The equality test and the relational order require the operators == and < for *T* respectively. These operators must not compare the pointers in *T*.

Types

In_place_list<T,bool>:: iterator
In_place_list<T,bool>:: const_iterator

In_place_list<T,bool>:: value_type
In_place_list<T,bool>:: reference
In_place_list<T,bool>:: const_reference
In_place_list<T,bool>:: size_type
In_place_list<T,bool>:: difference_type

In_place_list<T,bool>:: reverse_iterator
In_place_list<T,bool>:: const_reverse_iterator

In_place_list<T,bool>:: allocator_type

Creation

In_place_list<T,bool> l; — introduces an empty list.

In_place_list<T,bool> l(list<T> l1); — copy constructor. Each item in *l1* is copied.

In_place_list<T,bool> l(size_type n, T t = T());

introduces a list with n items, all initialized with copies of t.

template <class InputIterator>
In_place_list<T,bool> l(InputIterator first, InputIterator last);

a list with copies from the range [*first,last*).

In_place_list<T,bool> l(const T first, const T* last);*

non-member-template version.

In_place_list<T,bool> &	$l = l1$	assignment. Each item in *l1* is copied. Each item in *l* is deleted if the *bool* parameter is *true*.
void	*l.swap(l1)*	swaps the contents of *l* with *l1*.
void	*l.destroy()*	all items in *l* are deleted regardless of the *bool* parameter.

Comparison Operations

bool	$l == l1$	test for equality: Two lists are equal, iff they have the same size and if their corresponding elements are equal.
bool	$l < l1$	compares in lexicographical order.

Access Member Functions

iterator	*l.begin()*	returns a mutable iterator referring to the first element in *l*.
const_iterator	*l.begin() const*	returns a constant iterator referring to the first element in *l*.
iterator	*l.end()*	returns a mutable iterator which is the past-end-value of *l*.
const_iterator	*l.end() const*	returns a constant iterator which is the past-end-value of *l*.
bool	*l.empty()*	returns *true* if *l* is empty.
size_type	*l.size()*	returns the number of items in list *l*.
size_type	*l.max_size()*	returns the maximum possible size of the list *l*.
T&	*l.front()*	returns the first item in list *l*.
T&	*l.back()*	returns the last item in list *l*.

allocator_type

	l.get_allocator()	returns the allocator.

Insertion

void	*l.push_front(T&)*	inserts an item in front of list *l*.
void	*l.push_back(T&)*	inserts an item at the back of list *l*.

iterator	*l.insert(iterator pos, T& t)*	
iterator	*l.insert(T* pos, T& t)*	inserts *t* in front of *pos*. The return value points to the inserted item.

void *l.insert(iterator pos, size_type n, T t = T())*
void *l.insert(T* pos, size_type n, T t = T())*

inserts *n* copies of *t* in front of *pos*.

template <class InputIterator>
void *l.insert(iterator pos, InputIterator first, InputIterator last)*

template <class InputIterator>
void *l.insert(T* pos, InputIterator first, InputIterator last)*

inserts the range [*first, last*) in front of iterator *pos*.

As long as member templates are not supported, member functions using *T** instead of the general *InputIterator* are provided.

Removal

void	*l.pop_front()*	removes the first item from list *l*.
void	*l.pop_back()*	removes the last item from list *l*.
void	*l.erase(iterator pos)*	removes the item from list *l*, where *pos* refers to.
void	*l.erase(T* pos)*	removes the item from list *l*, where *pos* refers to.

void *l.erase(iterator first, iterator last)*
void *l.erase(T* first, T* last)*

removes the items in the range [*first, last*) from *l*.

Special List Operations

void	*l.splice(iterator pos, & x)*	
void	*l.splice(T* pos, & x)*	inserts the list *x* before position *pos* and *x* becomes empty. It takes constant time. *Precondition*: & *l*!= &*x*.

void *l.splice(iterator pos, & x, iterator i)* *void* *l.splice(T* pos, & x, T* i)*

inserts an element pointed to by i from list x before position *pos* and removes the element from x. It takes constant time. i is a valid dereferenceable iterator of x. The result is unchanged if $pos == i$ or $pos == ++i$.

void *l.splice(iterator pos, & x, iterator first, iterator last)*
void *l.splice(T* pos, & x, T* first, T* last)*

inserts elements in the range [*first, last*) before position *pos* and removes the elements from x. It takes constant time if $\&x == \&l$; otherwise, it takes linear time. [*first, last*) is a valid range in x.
Precondition: *pos* is not in the range [*first, last*).

void *l.remove(T value)* erases all elements e in the list l for which $e == value$. It is stable.
Precondition: a suitable *operator*== for the type T.

void *l.unique()* erases all but the first element from every consecutive group of equal elements in the list l.
Precondition: a suitable *operator*== for the type T.

void *l.merge(& x)* merges the list x into the list l and x becomes empty. It is stable.
Precondition: Both lists are increasingly sorted. A suitable *operator*< for the type T.

void *l.reverse()* reverses the order of the elements in l in linear time.

void *l.sort()* sorts the list l according to the *operator*< in time $O(n \log n)$ where n = *size()*. It is stable.
Precondition: a suitable *operator*< for the type T.

Example

```
#include <cassert>
#include <algorithm>
#include <CGAL/In_place_list.h>

using CGAL::In_place_list_base;

struct item : public In_place_list_base<item> {
    int key;
    item() {}
    item( const item& i) : In_place_list_base<item>(i), key(i.key) {}
    item( int i) : key(i) {}
    bool operator== (const item& i) const { return key == i.key;}
    bool operator!= (const item& i) const { return ! (*this == i);}
    bool operator== (int i) const         { return key == i;}
    bool operator!= (int i) const         { return ! (*this == i);}
    bool operator<  (const item& i) const { return key < i.key;}
};

int main() {
    typedef CGAL::In_place_list<item,true> List;
    List l;
    item* p = new item(1);
    l.push_back(*p);
    l.push_back(*new item(2));
    l.push_front(*new item(3));
    l.push_front(*new item(4));
    l.push_front(*new item(2));
    List::iterator i = l.begin();
    ++i;
    l.insert(i, *new item(5));
    l.insert(p, *new item(5));
    int a[7] = {2,5,4,3,5,1,2};
    bool ok = std::equal(l.begin(), l.end(), a);
    assert(ok);
    l.sort();
    l.unique();
    assert(l.size() == 5);
    int b[5] = {1,2,3,4,5};
    ok = std::equal(l.begin(), l.end(), b);
    assert(ok);
    return 0;
}
```

File: examples/STL_Extension/in_place_list_prog.cpp

CGAL::Compact_container_base

Class

Definition

The class *Compact_container_base* can be used as a base class for your own type *T*, so that *T* can be used directly within *Compact_container<T, Allocator>*. This class stores a *void* * pointer only for this purpose, so it may not be the most memory efficient way to achieve this goal. The other ways are to provide in *T* the necessary member functions so that the template *Compact_container_traits<T>* works, or to specialize it for the particular type *T* that you want to use.

#include <CGAL/Compact_container.h>

Operations

*void**	*ccb.for_compact_container() const*	Returns the pointer necessary for *Compact_container_traits<T>*.
*void**&	*ccb.for_compact_container()*	Returns a reference to the pointer necessary for *Compact_container_traits<T>*.

Class

CGAL::Compact_container_traits<T>

Definition

The traits class *Compact_container_traits<T>* provides the way to access the internal pointer required for *T* to be used in a *Compact_container<T, Allocator>*. Note that this pointer needs to be accessible even when the object is not constructed, which means it has to reside in the same memory place as *T*.

You can specialize this class for your own type *T* if the default template is not suitable.

You can also use *Compact_container_base* as base class for your own types *T* to make them usable with the default *Compact_container_traits<T>*.

#include <CGAL/Compact_container.h>

Parameters

T is any type providing the following member functions:
*void * t.for_compact_container() const;*
*void *& t.for_compact_container();*.

Operations

*static void**	*cct.pointer(const T &t)*	Returns the pointer hold by *t*. The template version defines this function as: *return t.for_compact_container();*
static void&*	*cct.pointer(T &t)*	Returns a reference to the pointer hold by *t*. The template version defines this function as: *return t.for_compact_container();*

CGAL::Compact_container<T, Allocator>

Class

Definition

An object of the class *Compact_container<T, Allocator>* is a container of objects of type T. It matches all the standard requirements for reversible containers, except that the complexity of its iterator increment and decrement operations is not always guaranteed to be amortized constant time.

This container is not a standard *sequence* nor *associative* container, which means the elements are stored in no particular order, and it is not possible to specify a particular place in the iterator sequence where to insert new objects. However, all dereferenceable iterators are still valid after calls to *insert()* and *erase()*, except those that have been erased (it behaves similarly to *std::list*).

The main feature of this container is that it is very memory efficient : its memory size is *N*sizeof(T)+o(N)*, where N is the maximum size that the container has had in its past history, its *capacity()* (the memory of erased elements is not deallocated until destruction of the container or a call to *clear()*). This container has been developed in order to store large graph-like data structures like the triangulation and the halfedge data structures.

It supports bidirectional iterators and allows a constant time amortized *insert()* operation. You cannot specify where to insert new objects (i.e. you don't know where they will end up in the iterator sequence, although *insert()* returns an iterator pointing to the newly inserted object). You can erase any element with a constant time complexity.

Summary of the differences with *std::list* : it is more compact in memory since it doesn't store two additional pointers for the iterator needs. It doesn't deallocate elements until the destruction or *clear()* of the container. The iterator does not have constant amortized time complexity for the increment and decrement operations in all cases, only when not too many elements have not been freed (i.e. when the *size()* is close to the *capacity()*). Iterating from *begin()* to *end()* takes *O(capacity())* time, not *size()*. In the case where the container has a small *size()* compared to its *capacity()*, we advise to "defragment the memory" by copying the container if the iterator performance is needed.

The iterators themselves can be used as T, they provide the necessary functions to be used by *Compact_container_traits<T>*. Moreover, they also provide a default constructor value which is not singular : it is copyable, comparable, and guaranteed to be unique under comparison (like *NULL* for pointers). This makes them suitable for use in geometric graphs like handles to vertices in triangulations.

In addition, in a way inspired from the Boost.Intrusive containers, it is possible to construct iterators from references to values in containers using the *iterator_to* and *s_iterator_to* functions.

#include <CGAL/Compact_container.h>

Parameters

The parameter T is required to have a copy constructor and an assignment operator. It also needs to provide access to an internal pointer via *Compact_container_traits<T>*.

The equality test and the relational order require the operators == and < for T respectively.

The parameter *Allocator* has to match the standard allocator requirements, with value type T. This parameter has the default value *CGAL_ALLOCATOR(T)*.

Types

Compact_container<T, Allocator>:: value_type
Compact_container<T, Allocator>:: reference
Compact_container<T, Allocator>:: const_reference
Compact_container<T, Allocator>:: pointer
Compact_container<T, Allocator>:: const_pointer
Compact_container<T, Allocator>:: size_type
Compact_container<T, Allocator>:: difference_type

Compact_container<T, Allocator>:: iterator
Compact_container<T, Allocator>:: const_iterator
Compact_container<T, Allocator>:: reverse_iterator
Compact_container<T, Allocator>:: const_reverse_iterator

Compact_container<T, Allocator>:: allocator_type

Creation

Compact_container<T, Allocator> c(Allocator a = Allocator());

introduces an empty container, eventually specifying a particular allocator *a* as well.

template <class InputIterator>
Compact_container<T, Allocator> c(InputIterator first, InputIterator last, Allocator a = Allocator());

a container with copies from the range [*first,last*), eventually specifying a particular allocator.

Compact_container<T, Allocator> c(cc);

copy constructor. Each item in *cc* is copied. The allocator is copied. The iterator order is preserved.

Compact_container<T, Allocator> & $c = cc$

assignment. Each item in *cc* is copied. The allocator is copied. Each item in *c* is deleted. The iterator order is preserved.

void *c.swap(&cc)*

swaps the contents of *c* and *cc* in constant time complexity. No exception is thrown.

void *c.reserve(size_type value)*

if *value* is less than or equal to *capacity()*, this call has no effect. Otherwise, it is a request for allocation of additional memory so that then *capacity()* is greater than or equal to value. *size()* is unchanged.

Access Member Functions

iterator	*c.begin()*	returns a mutable iterator referring to the first element in *c*.
const_iterator	*c.begin() const*	returns a constant iterator referring to the first element in *c*.
iterator	*c.end()*	returns a mutable iterator which is the past-end-value of *c*.
const_iterator	*c.end() const*	returns a constant iterator which is the past-end-value of *c*.
reverse_iterator	*c.rbegin()*	
const_reverse_iterator	*c.rbegin() const*	
reverse_iterator	*c.rend()*	
const_reverse_iterator	*c.rend() const*	
iterator	*c.iterator_to(reference value) const*	returns an iterator which points to *value*;
const_iterator	*c.iterator_to(const_reference value) const*	returns an iterator which points to *value*;
static iterator	*c.s_iterator_to(reference value)*	returns an iterator which points to *value*;
static const_iterator	*c.s_iterator_to(const_reference value)*	returns an iterator which points to *value*;
bool	*c.empty()*	returns *true* iff *c* is empty.
size_type	*c.size()*	returns the number of items in *c*.
size_type	*c.max_size()*	returns the maximum possible size of the container *c*.
size_type	*c.capacity()*	returns the total number of elements that *c* can hold without requiring reallocation.
Allocator	*c.get_allocator()*	returns the allocator.

Insertion

iterator *c.insert(T t)*

inserts a copy of *t* in *c* and returns the iterator pointing to it.

template <class InputIterator>
void *c.insert(InputIterator first, InputIterator last)*

inserts the range [*first, last*) in *c*.

template <class InputIterator>
void *c.assign(InputIterator first, InputIterator last)*

erases all the elements of *c*, then inserts the range [*first, last*) in *c*.

template < class T1 >
iterator *c.emplace(T1 t1)*

constructs an object of type *T* with the constructor that takes *t1* as argument, inserts it in *c*, and returns the iterator pointing to it. Overloads of this member function are defined that take additional arguments, up to 9.

Removal

void *c.erase(iterator pos)*

removes the item pointed by *pos* from *c*.

void *c.erase(iterator first, iterator last)*

removes the items from the range [*first, last*) from *c*.

void *c.clear()* all items in *c* are deleted, and the memory is deallocated. After this call, *c* is in the same state as if just default constructed.

Ownership testing

The following functions are mostly helpful for efficient debugging, since their complexity is $O(\sqrt{c.capacity()})$.

bool *c.owns(const_iterator pos)*

returns whether *pos* is in the range [*c.begin()*, *c.end()*] (*c.end()* included).

bool *c.owns_dereferencable(const_iterator pos)*

returns whether *pos* is in the range [*c.begin()*, *c.end()*) (*c.end()* excluded).

Merging

void	*c.merge(&cc)*	adds the items of *cc* to the end of *c* and *cc* becomes empty. The time complexity is O(*c.capacity()*-*c.size()*). *Precondition*: *cc* must not be the same as *c*, and the allocators of *c* and *cc* need to be compatible : *c.get_allocator() == cc.get_allocator()*.

Comparison Operations

bool	$c == cc$	test for equality: Two containers are equal, iff they have the same size and if their corresponding elements are equal.
bool	$c\ != cc$	test for inequality: returns !(*c* == *cc*).
bool	$c < cc$	compares in lexicographical order.
bool	$c > cc$	returns *cc* < *c*.
bool	$c <= cc$	returns !(*c* > *cc*).
bool	$c >= cc$	returns !(*c* < *cc*).

Class

CGAL::Multiset<Type,Compare,Allocator>

Definition

An instance *s* of the parametrized data type *Multiset<Type,Compare,Allocator>* is a multi-set of elements of type *Type*, represented as a red-black tree (see [CLRS01, Chapter 13] for an excellent introduction to red-black trees). The main difference between *Multiset<Type,Compare,Allocator>* and STL's *multiset* is that the latter uses a less-than functor with a Boolean return type, while our *Multiset<Type,Compare,Allocator>* class is parameterized by a comparison functor *Compare* that returns the three-valued *Comparison_result* (namely it returns either *SMALLER*, *EQUAL*, or *LARGER*). It is thus possible to maintain the underlying red-black tree with less invocations of the comparison functor. This leads to a speedup of about 5% even if we maintain a set of integers. When each comparison of two elements of type *Type* is an expensive operation (for example, when they are geometric entities represented using exact arithmetic), the usage of a three-valued comparison functor can lead to considerable decrease in the running times.

Moreover, *Multiset<Type,Compare,Allocator>* allows the insertion of an element into the set given its *exact* position, and not just using an insertion hint, as done by *std::multiset*. This can further reduce the running times, as additional comparison operations can be avoided.

In addition, the *Multiset<Type,Compare,Allocator>* guarantees that the order of elements sent to the comparison functor is fixed. For example, if we insert a new element *x* into the set (or erase an element from the set), then we always invoke *Compare() (x, y)* (and never *Compare() (y, x)*), where *y* is an element already stored in the set. This behavior, not supported by *std::multiset*, is sometimes crucial for designing more efficient comparison predicates.

Multiset<Type,Compare,Allocator> also allows for look-up of keys whose type may differ from *Type*, as long as users supply a comparison functor *CompareKey*, where *CompareKey() (key, y)* returns the three-valued *Comparison_result* (*key* is the look-up key and *y* is an element of type *Type*). Indeed, it is very convenient to look-up equivalent objects in the set given just by their key. We note however that it is also possible to use a key of type *Type* and to employ the default *Compare* functor for the look-up, as done when using the *std::multiset* class.

advanced

Finally, *Multiset<Type,Compare,Allocator>* introduces the *catenate()* and *split()* functions. The first function operates on *s* and accepts a second set *s*', such that the maximum element in *s* is not greater than the minimal element in *s*', and concatenates *s*' to *s*. The second function splits *s* into two sets,one containing all the elements that are less than a given key, and the other contains all elements greater than (or equal to) this key.

advanced

Parameters

The *Multiset* class-template has three parameters:

- *Type* — the type of the stored elements.
- *Compare* — the comparison-functor type. This type should provide the following operator for comparing two *Type* elements, namely:
  Comparison_result operator() (const Type& t1, const Type& t2) const;
  The *CGAL::Compare<Type>* functor is used by default. In this case, *Type* must support an equality operator (*operator==*) and a less-than operator (*operator<*).

- *Allocator* — the allocator type.
  CGAL_ALLOCATOR is used by default.

#include <CGAL/Multiset.h>

Assertions

The assertion and precondition flags for the *Multiset* class use *MULTISET* in their names (i.e., *CGAL_MULTISET_NO_ASSERTIONS* and *CGAL_MULTISET_NO_PRECONDITIONS*).

Types

In compliance with STL, the types *value_type* and *key_type* (both equivalent to *Type*), *reference* and *const_reference* (reference to a value-type), *key_compare* and *value_compare* (both equivalent to *Compare*), *size_type* and *difference_type* are defined as well.

Multiset<Type,Compare,Allocator>:: iterator
Multiset<Type,Compare,Allocator>:: const_iterator

bi-directional iterators for the elements stored in the set.

Multiset<Type,Compare,Allocator>:: reverse_iterator
Multiset<Type,Compare,Allocator>:: const_reverse_iterator

reverse bi-directional iterators for the elements stored in the set.

Creation

Multiset<Type,Compare,Allocator> s; creates an an empty set *s* that uses a default comparison functor.

Multiset<Type,Compare,Allocator> s(Compare comp);

creates an an empty set *s* that uses the given comparison functor *comp*.

template <class InputIterator>
Multiset<Type,Compare,Allocator> s(InputIterator first, InputIterator last, Compare comp = Compare());

creates a set *s* containing all elements in the range *[first, last)*, that uses the comparison functor *comp*.

Multiset<Type,Compare,Allocator> s(other); copy constructor.

Multiset<Type,Compare,Allocator>

$s = other$ assignment operator.

void	*s.swap(& other)*	swaps the contents of *s* with those of the other set.

Access Member Functions

Compare	*s.key_comp() const*	the comparison functor used.
Compare	*s.value_comp() const*	the comparison functor used (same as above). Both functions have a non-const version that return a reference to the comparison functor.
bool	*s.empty()*	returns *true* if the set is empty, *false* otherwise.
size_t	*s.size()*	returns the number of elements stored in the set.
size_t	*s.max_size()*	returns the maximal number of elements the set can store (same as *size()*).
iterator	*s.begin()*	returns an iterator pointing to the first element stored in the set (a *const* version is also available).
iterator	*s.end()*	returns an iterator pointing beyond the last element stored in the set (a *const* version is also available).
reverse_iterator	*s.rbegin()*	returns a reverse iterator pointing beyond the last element stored in the set (a *const* version is also available).
reverse_iterator	*s.rend()*	returns a reverse iterator pointing to the first element stored in the set (a *const* version is also available).

Comparison Operations

bool	*s == other*	returns *true* if the sequences of elements in the two sets are pairwise equal (using the comparison functor).
bool	*s < other*	returns *true* if the element sequence in *s* is lexicographically smaller than the element sequence of *other*.

Insertion Methods

iterator	*s.insert(Type x)*	inserts the element *x* into the set and returns an iterator pointing to the newly inserted element.

template <class InputIterator>
void *s.insert(InputIterator first, InputIterator last)*

inserts all elements in the range *[first, last)* into the set.

iterator *s.insert(iterator position, Type x)*

inserts the element x with a given iterator used as a hint for the position of the new element. It Returns an iterator pointing to the newly inserted element.

iterator *s.insert_before(iterator position, Type x)*

inserts the element x as the predecessor of the element at the given position.
Precondition: The operation does not violate the set order — that is, x is not greater than the element pointed by *position* and not less than its current predecessor.

iterator *s.insert_after(iterator position, Type x)*

inserts the element x as the successor of the element at the given position.
Precondition: The operation does not violate the set order — that is, x is not less than the element pointed by *position* and not greater than its current successor.

Removal Methods

size_t *s.erase(Type x)* erases all elements equivalent to x from the set and returns the number of erased elements.

void *s.erase(iterator position)*

erases the element pointed by *position*.

void *s.clear()* clears the set (erases all stored elements).

Look-up Methods

All methods listed in this section can also accept a *Type* element as a look-up key. In this case, it is not necessary to supply a *CompareKey* functor, as the *Compare* functor will be used by default.

template <class Key, class CompareKey>
iterator *s.find(Key key, CompareKey comp_key)*

searches for the an element equivalent to *key* in the set. If the set contains objects equivalent to *key*, it returns an iterator pointing to the first one. Otherwise, *end()* is returned (a *const* version is also available).

template <class Key, class CompareKey>

size_t *s.count(Key key, CompareKey comp_key) const*

returns the number of elements equivalent to *key* in the set.

template <class Key, class CompareKey>
iterator *s.lower_bound(Key key, CompareKey comp_key)*

returns an iterator pointing to the first element in the set that is not less than *key*. If all set elements are less than *key*, *end()* is returned (a *const* version is also available).

template <class Key, class CompareKey>
iterator *s.upper_bound(Key key, CompareKey comp_key)*

returns an iterator pointing to the first element in the set that is greater than *key*. If no set element is greater than *key*, *end()* is returned (a *const* version is also available).

template <class Key, class CompareKey>
std::pair<iterator,iterator>

s.equal_range(Key key, CompareKey comp_key)

returns the range of set elements equivalent to the given key, namely *(lower_bound(key), upper_bound(key))* (a *const* version is also available).

template <class Key, class CompareKey>
std::pair<iterator,bool>

s.find_lower(Key key, CompareKey comp_key)

returns a pair comprised of *lower_bound(key)* and a Boolean flag indicating whether this iterator points to an element equivalent to the given key (a *const* version is also available).

advanced

Special Operations

void *s.replace(iterator position, Type x)*

replaces the element stored at the given position with *x*.
Precondition: The operation does not violate the set order — that is, *x* is not less that *position*'s predecessor and not greater than its successor.

void *s.swap(iterator pos1, iterator pos2)*

swaps places between the two elements given by *pos1* and *pos2*.
Precondition: The operation does not violate the set order — that is, *pos1* and *pos2* store equivalent elements.

void *s.catenate(Self& s_prime)*

concatenates all elements in *s_prime* into *s* and clears *s_prime*. All iterators to *s* and to *s_prime* remain valid.
Precondition: The maximal element in *s* is not greater than the minimal element in *s_prime*.

template <class Key, class CompareKey>
void *s.split(Key key, CompareKey comp_key, Self& s_prime)*

splits *s* such that it contains all elements that are less than the given *key* and such that *s_prime* contains all other elements.
Precondition: *s_prime* is initially empty.

void *s.split(iterator position, Self& s_prime)*

splits *s* such that it contains all set elements in the range *[begin, position)* and such that *s_prime* contains all elements in the range *[position, end())*.
Precondition: *s_prime* is initially empty.

advanced

Implementation

Multiset uses a proprietary implementation of a red-black tree data-structure. The red-black tree invariants guarantee that the height of a tree containing n elements is $O(\log n)$ (more precisely, it is bounded by $2\log_2 n$). As a consequence, all methods that accept an element and need to locate it in the tree (namely *insert(x)*, *erase(x)*, *find(x)*, *count(x)*, *lower_bound(x)* , *upper_bound(x)*, *find_lower(x)* and *equal_range(x)*) take $O(\log n)$ time and perform $O(\log n)$ comparison operations.

On the other hand, the set operations that accept a position iterator (namely *insert_before(pos, x)*, *insert_after(pos, x)* and *erase(pos)*) are much more efficient as they can be performed at a *constant* amortized cost (see [GS78] and [Tar83] for more details). More important, these set operations require *no* comparison operations. Therefore, it is highly recommended to maintain the set via iterators to the stored elements, whenever possible. The function *insert(pos, x)* is safer to use, but it takes amortized $O(\min\{d, \log n\})$ time, where d is the distance between the given position and the true position of x. In addition, it always performs at least two comparison operations.

advanced

The *catenate()* and *split()* functions are also very efficient, and can be performed in $O(\log n)$ time, where n is the total number of elements in the sets, and without performing any comparison operations (see [Tar83] for the details). Note however that the size of two sets resulting from a split operation is initially unknown, as it is impossible to compute it in less than linear time. Thus, the first invocation of *size()* on such a set takes linear time, and *not* constant time.

advanced

The design is derived from the STL *multiset* class-template (see, e.g, [MS96]), where the main differences between the two classes are highlighted in the class definition above.

CGAL::Object

#include <CGAL/Object.h>

Definition

Some functions can return different types of objects. A typical C++ solution to this problem is to derive all possible return types from a common base class, to return a pointer to this class and to perform a dynamic cast on this pointer. The class *Object* provides an abstraction. An object *obj* of the class *Object* can represent an arbitrary class. The only operations it provides is to make copies and assignments, so that you can put them in lists or arrays. Note that *Object* is NOT a common base class for the elementary classes. Therefore, there is no automatic conversion from these classes to *Object*. Rather this is done with the global function *make_object*. This encapsulation mechanism requires the use of *assign* or *object_cast* to use the functionality of the encapsulated class.

This class is similar in spirit to *boost::any*.

Creation

Object obj;	introduces an empty object.
Object obj(o);	Copy constructor.

Objects of type *Object* are normally created using the global function *make_object*.

Operations

Object &	*obj* = *o*	Assignment.
bool	*obj.empty()*	returns true, if *obj* does not contain an object.
template<class T> *bool*	*obj.is()*	returns true, iff *obj* contains an object of type *T*.
std::type_info	*obj.type() const*	returns the type information of the contained type, or *typeid(void)* if empty.

Construction of an *Object* storing an object of type *T* can be performed using the *make_object* global function :

template <class T> *Object*	*make_object(T t)*	Creates an object that contains *t*.

Assignment of an object of type *Object* to an object of type *T* can be done using *assign* :

template <class T> *bool*	*assign(T& c, o)*	assigns *o* to *c* if *o* was constructed from an object of type *T*. Returns *true*, if the assignment was possible. For efficiency reasons, we recommend using *object_cast* instead.

Another possibility to access the encapsulated object is to use *object_cast*, which avoids the default constructor and assignment required by *assign* :

template <class T>
*const T** *object_cast(const * o)*

Returns a pointer to the object of type *T* stored by *o*, if any, otherwise returns *NULL*.

template <class T>
T *object_cast(o)*

Returns a copy of the object of type *T* stored by *o*, if any, otherwise throws an exception of type *Bad_object_cast*.

Example

In the following example, the object class is used as return value for the intersection computation, as there are possibly different return values.

```
{
    typedef Cartesian<double>    K;
    typedef K::Point_2           Point_2;
    typedef K::Segment_2         Segment_2;

    Point_2 point;
    Segment_2 segment, segment_1, segment_2;

    std::cin >> segment_1 >> segment_2;

    Object obj = intersection(segment_1, segment_2);

    if (assign(point, obj)) {
        /* do something with point */
    } else if (assign(segment, obj)) {
        /* do something with segment*/
    }

    /* there was no intersection */
}
```

A more efficient way to access the object is to use *object_cast*, which allows to skip a default construction and assignment :

```
{
    typedef Cartesian<double>    K;
    typedef K::Point_2           Point_2;
    typedef K::Segment_2         Segment_2;

    Segment_2 segment_1, segment_2;

    std::cin >> segment_1 >> segment_2;
```

```
    Object obj = intersection(segment_1, segment_2);

    if (const Point_2 * point = object_cast<Point_2>(&obj)) {
        /* do something with *point */
    } else if (const Segment_2 * segment = object_cast<Segment_2>(&obj)) {
        /* do something with *segment*/
    }

    /* there was no intersection */
}
```

The intersection routine itself looks roughly as follows:

```
template < class Kernel >
Object intersection(Segment_2<Kernel> s1, Segment_2<Kernel> s2)
{

    if (/* intersection is a point */ ) {

      Point_2<Kernel> p = ...  ;
      return make_object(p);

    } else if (/* intersection is a segment */ ) {

      Segment_2<Kernel> s = ...  ;
      return make_object(s);
    }

    /* empty intersection */
    return Object();
}
```

Function

CGAL::cpp0x::next

There is actually no function in namespace *CGAL::cpp0x* with this name, but a using declaration which imports a function from another namespace. By order of priority: the one in namespace *std* is used (provided by C++0x), if not found, then the one in namespace *boost* is used.

Definition

The function returns the result of *operator++* on a ForwardIterator. The exact behaviour is described in §24.4.4 of the C++ standard draft N3242.

#include <CGAL/algorithm.h>

See Also

boost::next
CGAL::cpp0x::prev . . . . page 4171

CGAL::cpp0x::prev

Function

If C++0x is available the function *std::prev* is imported into the namespace *CGAL::cpp0x*, otherwise *CGAL::cpp0x::prev* is declared with the signature as given in §24.4.4 of the ISO C++ Standard and forwarded to *boost::prior*.

Definition

The function returns the result of *operator--* on a BidirectionalIterator. The exact behaviour is described in §24.4.4 of the C++ standard draft N3242.

#include <CGAL/algorithm.h>

See Also

boost::prior
CGAL::cpp0x::next . . . . page 4170

CGAL::predecessor

This function is deprecated. *CGAL::cpp0x::prev* should be used instead.

deprecated

Definition

The function *predecessor* returns the previous iterator, i.e. the result of *operator--* on a bidirectional iterator.

#include <CGAL/algorithm.h>

template <class BidirectionalIterator>
BidirectionalIterator predecessor(BidirectionalIterator it)

returns *--it*.

See Also

CGAL::successor .... page 4173

deprecated

CGAL::successor

This function is deprecated. *CGAL::cpp0x::next* should be used instead.

deprecated

Definition

The function *successor* returns the next iterator, i.e. the result of *operator++* on a forward iterator.

#include <CGAL/algorithm.h>

template <class ForwardIterator>
ForwardIterator successor(ForwardIterator it)

returns $++it$.

See Also

CGAL::predecessor .... page 4172

deprecated

Function

CGAL::cpp0x::copy_n

This provides an implementation of the standard function *copy_n* from the C++0x standard. If *copy_n* is available in the *std::* namespace a using declaration is used, otherwise an alternative implementation from CGAL is used.

Definition

The function *cpp0x::copy_n* copies *n* items from an input iterator to an output iterator. Its exact behaviour is defined in §25.3.1 of the C++ standard draft N3242.

#include <CGAL/algorithm.h>

CGAL::copy_n

Function

This function is deprecated, *CGAL::cpp0x::copy_n* should be used instead.

deprecated

Definition

The function *copy_n* copies *n* items from an input iterator to an output iterator which is useful for possibly infinite sequences of random geometric objects.[1]

#include <CGAL/algorithm.h>

template <class InputIterator, class Size, class OutputIterator>
OutputIterator copy_n(InputIterator first, Size n, OutputIterator result)

copies the first *n* items from *first* to *result*. Returns the value of *result* after inserting the *n* items.

See Also

CGAL::Counting_iterator<Iterator, Value> . . . . page 4185

deprecated

[1]The STL release June 13, 1997, from SGI contains an equivalent function, but it is not part of the ISO standard.

Function

CGAL::min_max_element

Definition

The function *min_max_element* computes the minimal and the maximal element of a range. It is modeled after the STL functions *min_element* and *max_element*. The advantage of *min_max_element* compared to calling both STL functions is that one only iterates once over the sequence. This is more efficient especially for large and/or complex sequences.

#include <CGAL/algorithm.h>

template < class ForwardIterator >
std::pair< ForwardIterator, ForwardIterator >

min_max_element(ForwardIterator first, ForwardIterator last)

returns a pair of iterators where the first component refers to the minimal and the second component refers to the maximal element in the range [*first*, *last*). The ordering is defined by *operator<* on the value type of *ForwardIterator*.

template < class ForwardIterator, class CompareMin, class CompareMax >
std::pair< ForwardIterator, ForwardIterator >

min_max_element(ForwardIterator first,
ForwardIterator last,
CompareMin comp_min,
CompareMax comp_max)

returns a pair of iterators where the first component refers to the minimal and the second component refers to the maximal element in the range [*first*, *last*).
Requirement: *CompareMin* and *CompareMax* are adaptable binary function objects: $VT \times VT \rightarrow bool$ where *VT* is the value type of *ForwardIterator*.

Example

The following example program computes the minimal and maximal element of the sequence $(3, 6, 5)$. Hence the output is *min = 3, max = 6*.

```
#include <CGAL/algorithm.h>
#include <vector>
#include <iostream>

using std::vector;
using std::pair;
using std::cout;
using std::endl;
using CGAL::min_max_element;
```

```
int main()
{
  vector< int > v;
  v.push_back(3);
  v.push_back(6);
  v.push_back(5);
  typedef std::vector< int >::iterator iterator;
  pair< iterator, iterator > p = min_max_element(v.begin(), v.end());
  cout << "min = " << *p.first << ", max = " << *p.second << endl;
  return 0;
}
```

File: examples/STL_Extension/min_max_element_example.cpp

Class

CGAL::Dispatch_output_iterator<V,O>

Definition

The class *Dispatch_output_iterator<V,O>* defines an *OutputIterator* that contains a tuple of output iterators, and dispatches among those based on the type of the value type which is put in it. It also inherits from *O*, which makes it easy to treat like a tuple.

Parameters

V must be a *CGAL::cpp0x::tuple<...>* of the types of values to be accepted and dispatched. *O* must be a *CGAL::cpp0x::tuple<...>* of the types of corresponding output iterators.

#include <CGAL/iterator.h>

Is Model for the Concepts

OutputIterator

Inherits From

O

Types

typedef V	*Value_type_tuple;*
typedef O	*Iterator_tuple;*

Creation

Dispatch_output_iterator<V,O> i(I...o);

Constructor taking all the output iterators.

Iterator_tuple *i.get_iterator_tuple() const*

returns a reference to the tuple of output iterators.

template < typename... V, typename... O>
Dispatch_output_iterator<tuple<V...>, tuple<O...> >

dispatch_output(O... o)

returns a *Dispatch_output_iterator* constructed from the arguments.

See Also

CGAL::Dispatch_or_drop_output_iterator<V,O> . . . . page 4180

Class

CGAL::Dispatch_or_drop_output_iterator<V,O>

Definition

The class *Dispatch_or_drop_output_iterator<V,O>* defines an *OutputIterator* that contains a tuple of output iterators, and dispatches among those based on the type of the value type which is put in it. Other types are also accepted, and the object is simply discarded in this case. It also inherits from *O*, which makes it easy to treat like a tuple.

Parameters

V must be a *CGAL::cpp0x::tuple<...>* of the types of values to be accepted and dispatched. *O* must be a *CGAL::cpp0x::tuple<...>* of the types of corresponding output iterators.

#include <CGAL/iterator.h>

Is Model for the Concepts

OutputIterator

Inherits From

O

Types

typedef V	*Value_type_tuple;*
typedef O	*Iterator_tuple;*

Creation

Dispatch_or_drop_output_iterator<V,O> i(I...o);

Constructor taking all the output iterators.

Iterator_tuple *i.get_iterator_tuple() const*

returns a reference to the tuple of output iterators.

template < typename... V, typename... O>
Dispatch_or_drop_output_iterator<tuple<V...>, tuple<O...> >

dispatch_or_drop_output(O... o)

returns a *Dispatch_or_drop_output_iterator* constructed from the arguments.

See Also

CGAL::Dispatch_output_iterator<V,O> . . . . page 4178

Class

CGAL::Emptyset_iterator

Definition

The class defines an *OutputIterator* that ignores everything written to it. One can think of it as being connected to `/dev/null`.

#include <CGAL/iterator.h>

Is Model for the Concepts

OutputIterator

Creation

Emptyset_iterator i;

default constructor.

See Also

CGAL::Oneset_iterator . . . . page **??**
CGAL::Const_oneset_iterator . . . . page **??**

CGAL::Oneset_iterator<T>

Class

Definition

The class *Oneset_iterator<T>* defines an *BidirectionalIterator* that always refers to one specific object of type *T*. Internally, *Oneset_iterator<T>* stores a pointer to the referred object.

#include <CGAL/iterator.h>

Is Model for the Concepts

BidirectionalIterator

Creation

Oneset_iterator<T> i(T& t);

creates an iterator referring to *t*.

See Also

CGAL::Emptyset_iterator .... page 4182
CGAL::Const_oneset_iterator .... page **??**

Class

CGAL::Const_oneset_iterator<T>

Definition

The class *Const_oneset_iterator<T>* defines an *RandomAccessIterator* that always refers to a copy of a specific object of type T.

#include <CGAL/iterator.h>

Is Model for the Concepts

RandomAccessIterator

Creation

Const_oneset_iterator<T> i(T& t);

creates an iterator that always refers to some copy of t. The copy is constructed by invoking T's copy constructor and remains constant during i's lifetime.

See Also

CGAL::Emptyset_iterator . . . . page 4182
CGAL::Oneset_iterator . . . . page **??**

CGAL::Counting_iterator<Iterator, Value>

Definition

The iterator adaptor *Counting_iterator<Iterator, Value>* adds a counter to the internal iterator of type *Iterator* and defines equality of two instances in terms of this counter. It can be used to create finite sequences of possibly infinite sequences of values from input iterators.

#include <CGAL/iterator.h>

Is Model for the Concepts

InputIterator

Requirements

Iterator is a model for *InputIterator*.

Creation

Counting_iterator<Iterator, Value> i(std::size_t n = 0);

initializes the internal counter to n and i has a singular value.

Counting_iterator<Iterator, Value> i(Iterator j, std::size_t n = 0);

initializes the internal counter to n and i to j.

See Also

CGAL::copy_n . . . . page 4175

Class

CGAL::Insert_iterator<Container>

Definition

The output iterator *Insert_iterator<Container>* is similar to *std::insert_iterator*, but differs in that it calls the *insert()* function of the container without the iterator additional argument.

#include <CGAL/iterator.h>

Is Model for the Concepts

OutputIterator

Requirements

Container provides a member function *insert(const Container::const_reference&)*.

Creation

Insert_iterator<Container> i(Container &c);

initializes the internal container reference to *c*.

There is also a global function similar to *std::inserter*:

template < class Container >
Insert_iterator<Container>

inserter(Container &x)

Constructs *Insert_iterator<Container>(x)*.

CGAL::N_step_adaptor<I,int N>

Class

Definition

The adaptor *N_step_adaptor<I,int N>* changes the step width of the iterator or circulator class I to N. It is itself an iterator or circulator respectively. The behavior is undefined if the adaptor is used on a range $[i, j)$ where $j - i$ is not a multiple of n.

#include <CGAL/iterator.h>

Creation

N_step_adaptor<I,int N> i(I j);	down cast.

Class

CGAL::Filter_iterator<Iterator, Predicate>

Definition

The iterator adaptor *Filter_iterator<Iterator, Predicate>* acts as a filter on a given range. Whenever the iterator is in– or decremented, it ignores all iterators for which the given *Predicate* is true. The iterator category is the same as for *Iterator*.

Note: Boost also provides the same functionality via the *boost::filter_iterator* class. Unfortunately, the semantics chosen for accepting or rejecting elements based on the predicate's result are opposite as the semantic chosen here. What is more, the argument of the predicate is different: the predicate used with *boost::filter_iterator* must take the value type of the iterator, as argument, and not the iterator itself.

#include <CGAL/iterator.h>

Requirements

- *Iterator* is a model for *ForwardIterator*.
- *Predicate* is a functor: *Iterator* $\rightarrow$ *bool*.

Creation

Filter_iterator<Iterator, Predicate> *i*;

Filter_iterator<Iterator, Predicate> *i(Iterator e, Predicate p, Iterator c = e);*

creates an iterator which filters values according to *p*. Initializes by taking the first valid iterator (according to *p*), starting at *c*, and stopping at *e* if none is found.

There is also a global function to help the use of *Filter_iterator<Iterator, Predicate>*:

template < class Iterator, class Predicate >
inline Filter_iterator< Iterator, Predicate >

filter_iterator(Iterator e, Predicate p, Iterator c = e)

Constructs *Filter_iterator<Iterator, Predicate>(e, p, c)*.

CGAL::Join_input_iterator_1<Iterator, Creator>

Class

Definition

The class *Join_input_iterator_1<Iterator, Creator>* joins an iterator and a creator function object. The result is again an iterator (of the same iterator category type as the original iterator) that reads an object from the stream and applies a creator function object to that object.

#include <CGAL/iterator.h>

Is Model for the Concepts

InputIterator

Types

Join_input_iterator_1<Iterator, Creator>:: value_type

typedef to *Creator::result_type*.

Creation

Join_input_iterator_1<Iterator, Creator> join(Iterator i, Creator creator);

creates a join iterator from the given iterator *i* and the functor *creator*. Applies *creator* to each item read from *i*.

Join_input_iterator_1<Iterator, Creator> join(Iterator i);

creates a join iterator from the given iterator *i* and a default constructed instance of *Creator*. The latter instance is applied to each item read from *i*.

See Also

CGAL::Creator_1<Arg, Result> . . . . page 4209

CGAL::Inverse_index<IC>

Definition

The class *Inverse_index<IC>* constructs an inverse index for a given range $[i, j)$ of two iterators or circulators of type *IC*. The first element *I* in the range $[i, j)$ has the index 0. Consecutive elements are numbered incrementally. The inverse index provides a query for a given iterator or circulator k to retrieve its index number. *Precondition:* The iterator or circulator must be either of the random access category or the dereference operator must return stable and distinguishable addresses for the values, e.g. proxies or non-modifiable iterator with opaque values will not work.

#include <CGAL/iterator.h>

Creation

Inverse_index<IC>	*inverse*;	invalid index.
Inverse_index<IC>	*inverse(IC i);*	empty inverse index initialized to start at i.
Inverse_index<IC>	*inverse(IC i, IC j);*	inverse index initialized with range $[i, j)$.

Operations

std::size_t	*inverse*[*IC k*]	returns inverse index of k. *Precondition*: k has been stored in the inverse index.
void	*inverse.push_back(IC k)*	adds k at the end of the indices.

Implementation

For random access iterators or circulators, it is done in constant time by subtracting i. For other iterator categories, an STL *map* is used, which results in a $\log j - i$ query time. The comparisons are done using the operator *operator<* on pointers.

See Also

CGAL::Random_access_adaptor<IC> . . . . page 4191
CGAL::Random_access_value_adaptor<IC,T> . . . . page 4192

CGAL::Random_access_adaptor<IC>

Class

Definition

The class *Random_access_adaptor<IC>* provides a random access for data structures. Either the data structure supports random access iterators or circulators where this class maps function calls to the iterator or circulator, or a STL *std::vector* is used to provide the random access. The iterator or circulator of the data structure are of type *IC*.

#include <CGAL/iterator.h>

Types

Random_access_adaptor<IC>:: size_type	size type of the STL *std::vector*.

Creation

Random_access_adaptor<IC> random_access; invalid index.

Random_access_adaptor<IC> random_access(IC i);

empty random access index initialized to start at *i*.

Random_access_adaptor<IC> random_access(IC i, IC j);

random access index initialized to the range $[i, j)$.

void

random_access.reserve(size_type r)

reserve *r* entries, if a *std::vector* is used internally.

Operations

IC *random_access*[*size_type n*] returns iterator or circulator to the *n*-th item.
Precondition: $n <$ number of items in *random_access*.

void

random_access.push_back(IC k) adds *k* at the end of the indices.

See Also

CGAL::Inverse_index<IC> . . . . page 4190
CGAL::Random_access_value_adaptor<IC,T> . . . . page 4192

Class

CGAL::Random_access_value_adaptor<IC,T>

Definition

The class *Random_access_value_adaptor<IC,T>* provides a random access for data structures. It is derived from *Random_access_adaptor<IC>*. Instead of returning iterators from the *operator[]* methods, it returns the dereferenced value of the iterator. The iterator or circulator of the data structure are of type *IC*. Their value type is T.

#include <CGAL/iterator.h>

Operations

Creation and operations see *Random_access_adaptor<IC>*, with the exception of:

T&	*random_access*[*size_type n*]	returns a reference to the n-th item. *Precondition*: $n <$ number of items in *random_access*.

See Also

CGAL::Inverse_index<IC> . . . . page 4190
CGAL::Random_access_adaptor<IC> . . . . page 4191

CGAL::compare_to_less

Definition

The function *compare_to_less* is used to change a functor returning a *Comparison_result* to one which returns a bool. The returned functor will return *true* iff the original one returns *SMALLER*.

#include <CGAL/function_objects.h>

template < class F >
Compare_to_less< F >   *compare_to_less(F f)*

returns a functor equivalent to f, but which returns a bool instead of a *Comparison_result*.

See Also

CGAL::Compare_to_less<F> . . . . page 4194

Class

CGAL::Compare_to_less<F>

Definition

The class *Compare_to_less<F>* is used to convert a functor which returns a *Comparison_result* to a predicate (returning bool) : it will return true iff the return value of *F* is *SMALLER*. The class is used in conjunction with the *compare_to_less* function; see there for an explanation on how exactly the functors are combined.

#include <CGAL/function_objects.h>

Types

Compare_to_less<F>:: Type	type of the composed functor.

See Also

CGAL::compare_to_less . . . . page 4193

Projection_object

Functor
Concept

Definition

The concept Projection_object is modeled after the STL concept *UnaryFunction*, but takes also care of (const) references.

Projection_object:: argument_type	argument type.
Projection_object:: result_type	result type.

Creation

Projection_object o;	default constructor.

Operations

result_type`&`	*o.operator()(argument_type* `&`*) const*
const result_type`&`	*o.operator()(const argument_type* `&`*) const*

Has Models

Functor
Class

CGAL::Identity<Value>

Definition

The class *Identity<Value>* represents the identity function on *Value*.

#include <CGAL/function_objects.h>

Is Model for the Concepts

Projection_object . . . . page 4195

Identity<Value>:: argument_type	typedef to *Value*.
Identity<Value>:: result_type	typedef to *Value*.

Creation

Identity<Value> o;	default constructor.

Operations

result_type&	*o.operator()(argument_type& x) const*
	returns *x*.
const result_type&	*o.operator()(const argument_type& x) const*
	returns *x*.

CGAL::Dereference<Value>

Definition

The class *Dereference<Value>* dereferences a pointer (*operator**).

#include <CGAL/function_objects.h>

Is Model for the Concepts

Projection_object . . . . page 4195

Dereference<Value>:: argument_type	typedef to *Value**.
Dereference<Value>:: result_type	typedef to *Value*.

Creation

Dereference<Value> o;	default constructor.

Operations

result_type&   *o.operator()(argument_type& x) const*

returns **x*.

const result_type&   *o.operator()(const argument_type& x) const*

returns **x*.

CGAL::Get_address<Value>

Definition

The class *Get_address<Value>* gets the address of an lvalue (*operator*`&`).

#include <CGAL/function_objects.h>

Is Model for the Concepts

Projection_object . . . . page 4195

Get_address<Value>:: argument_type typedef to *Value*.

Get_address<Value>:: result_type typedef to *Value**.

Creation

Get_address<Value> o; default constructor.

Operations

result_type`&` *o.operator()(argument_type*`&` *x) const*

returns `&`*x*.

const result_type`&` *o.operator()(const argument_type*`&` *x) const*

returns `&`*x*.

CGAL::Cast_function_object<Arg, Result>

Functor
Class

Definition

The class *Cast_function_object<Arg, Result>* applies a C-style type cast to its argument.

#include <CGAL/function_objects.h>

Is Model for the Concepts

Projection_object . . . . page 4195

Cast_function_object<Arg, Result>:: argument_type

typedef to *Arg*.

Cast_function_object<Arg, Result>:: result_type

typedef to *Result*.

Creation

Cast_function_object<Arg, Result> o; default constructor.

Operations

result_type& *o.operator()(argument_type*& *x) const*

returns *(Result)x*.

const result_type& *o.operator()(const argument_type*& *x) const*

returns *(Result)x*.

Functor
Class

CGAL::Project_vertex<Node>

Definition

The class *Project_vertex<Node>* calls the member function *vertex()* on an instance of type *Node*.

#include <CGAL/function_objects.h>

Is Model for the Concepts

Projection_object . . . . page 4195

Project_vertex<Node>:: argument_type	typedef to *Node*.
Project_vertex<Node>:: result_type	typedef to *Node::Vertex*.

Creation

Project_vertex<Node> o;	default constructor.

Operations

result_type& *o.operator()(argument_type& n) const*

returns *n.vertex()*.

const result_type& *o.operator()(const argument_type& n) const*

returns *n.vertex()*.

CGAL::Project_facet<Node>

Definition

The class *Project_facet<Node>* calls the member function *facet()* on an instance of type *Node*.

#include <CGAL/function_objects.h>

Is Model for the Concepts

Projection_object . . . . page 4195

Project_facet<Node>:: argument_type	typedef to *Node*.
Project_facet<Node>:: result_type	typedef to *Node::Facet*.

Creation

Project_facet<Node> o;	default constructor.

Operations

result_type& *o.operator()(argument_type& n) const*

returns *n.facet()*.

const result_type& *o.operator()(const argument_type& n) const*

returns *n.facet()*.

Functor
Class

CGAL::Project_point<Node>

Definition

The class *Project_point<Node>* calls the member function *point()* on an instance of type *Node*.

#include <CGAL/function_objects.h>

Is Model for the Concepts

Projection_object . . . . page 4195

Project_point<Node>:: argument_type	typedef to *Node*.
Project_point<Node>:: result_type	typedef to *Node::Point*.

Creation

Project_point<Node> o;	default constructor.

Operations

result_type& *o.operator()(argument_type& n) const*

returns *n.point()*.

const result_type& *o.operator()(const argument_type& n) const*

returns *n.point()*.

CGAL::Project_normal<Node>

Definition

The class *Project_normal<Node>* calls the member function *normal()* on an instance of type *Node*.

#include <CGAL/function_objects.h>

Is Model for the Concepts

Projection_object . . . . page 4195

Project_normal<Node>:: argument_type	typedef to *Node*.
Project_normal<Node>:: result_type	typedef to *Node::Normal*.

Creation

Project_normal<Node> o;	default constructor.

Operations

result_type& *o.operator()(argument_type& n) const*

returns *n.normal()*.

const result_type& *o.operator()(const argument_type& n) const*

returns *n.normal()*.

CGAL::Project_plane<Node>

Definition

The class *Project_plane<Node>* calls the member function *plane()* on an instance of type *Node*.

#include <CGAL/function_objects.h>

Is Model for the Concepts

Projection_object . . . . page 4195

Project_plane<Node>:: argument_type	typedef to *Node*.
Project_plane<Node>:: result_type	typedef to *Node::Plane*.

Creation

Project_plane<Node> o;	default constructor.

Operations

result_type& *o.operator()(argument_type& n) const*

returns *n.plane()*.

const result_type& *o.operator()(const argument_type& n) const*

returns *n.plane()*.

CGAL::Project_next<Node>

Definition

The class *Project_next<Node>* calls the member function *next()* on an instance of type *Node*.

#include <CGAL/function_objects.h>

Is Model for the Concepts

Projection_object . . . . page 4195

Project_next<Node>:: argument_type	typedef to *Node**.
Project_next<Node>:: result_type	typedef to *Node**.

Creation

Project_next<Node> o;	default constructor.

Operations

result_type&	*o.operator()(argument_type& n) const*
	returns *n->next()*.
const result_type&	*o.operator()(const argument_type& n) const*
	returns *n->next()*.

Functor Class

Functor
Class

CGAL::Project_prev<Node>

Definition

The class *Project_prev<Node>* calls the member function *prev()* on an instance of type *Node*.

#include <CGAL/function_objects.h>

Is Model for the Concepts

Projection_object . . . . page 4195

Project_prev<Node>:: argument_type	typedef to *Node**.
Project_prev<Node>:: result_type	typedef to *Node**.

Creation

Project_prev<Node> o;	default constructor.

Operations

result_type&	*o.operator()(argument_type& n) const*
	returns *n->prev()*.
const result_type&	*o.operator()(const argument_type& n) const*
	returns *n->prev()*.

Functor
Class

CGAL::Project_next_opposite<Node>

Definition

The class *Project_next_opposite<Node>* calls the member functions *next()->opposite()* on an instance of type *Node*.

#include <CGAL/function_objects.h>

Is Model for the Concepts

Projection_object . . . . page 4195

Project_next_opposite<Node>:: argument_type	typedef to *Node**.
Project_next_opposite<Node>:: result_type	typedef to *Node**.

Creation

Project_next_opposite<Node> o;	default constructor.

Operations

result_type& *o.operator()(argument_type& n) const*

returns *n->next()->opposite()*.

const result_type& *o.operator()(const argument_type& n) const*

returns *n->next()->opposite()*.

Functor
Class

CGAL::Project_opposite_prev<Node>

Definition

The class *Project_opposite_prev<Node>* calls the member functions *opposite()->prev()* on an instance of type *Node*.

#include <CGAL/function_objects.h>

Is Model for the Concepts

Projection_object . . . . page 4195

Project_opposite_prev<Node>:: argument_type

typedef to *Node**.

Project_opposite_prev<Node>:: result_type typedef to *Node**.

Creation

Project_opposite_prev<Node> o; default constructor.

Operations

result_type& *o.operator()(argument_type& n) const*

returns *n->opposite()->prev()*.

const result_type& *o.operator()(const argument_type& n) const*

returns *n->opposite()->prev()*.

CGAL::Creator_1<Arg, Result>

Definition

The concept *Creator_1<Arg, Result>* defines types and operations for creating objects from one argument.

#include <CGAL/function_objects.h>

Requirements

Arg is convertible to *Result*.

Creator_1<Arg, Result>:: argument_type	type of argument.
Creator_1<Arg, Result>:: result_type	type of object to create.

result_type *c.operator()(argument_type a) const*

returns *result_type(a)*.

Functor
Class

CGAL::Creator_2<Arg1, Arg2, Result>

Definition

The concept *Creator_2<Arg1, Arg2, Result>* defines types and operations for creating objects from two arguments.

#include <CGAL/function_objects.h>

Requirements

Result defines a corresponding constructor.

Creator_2<Arg1, Arg2, Result>:: argument1_type

type of first argument.

Creator_2<Arg1, Arg2, Result>:: argument2_type

type of second argument.

Creator_2<Arg1, Arg2, Result>:: result_type type of object to create.

result_type *c.operator()(argument_type1 a1, argument_type2 a2) const*

returns *result_type(a1, a2)*.

CGAL::Creator_3<Arg1, Arg2, Arg3, Result>

Functor
Class

Definition

The concept *Creator_3<Arg1, Arg2, Arg3, Result>* defines types and operations for creating objects from three arguments.

#include <CGAL/function_objects.h>

Requirements

Result defines a corresponding constructor.

Creator_3<Arg1, Arg2, Arg3, Result>:: argument1_type

type of first argument.

Creator_3<Arg1, Arg2, Arg3, Result>:: argument2_type

type of second argument.

Creator_3<Arg1, Arg2, Arg3, Result>:: argument3_type

type of third argument.

Creator_3<Arg1, Arg2, Arg3, Result>:: result_type

type of object to create.

result_type *c.operator()(argument_type1 a1, argument_type2 a2, argument_type3 a3) const*

returns *result_type(a1, a2, a3)*.

Functor
Class

CGAL::Creator_4<Arg1, Arg2, Arg3, Arg4, Result>

Definition

The concept *Creator_4<Arg1, Arg2, Arg3, Arg4, Result>* defines types and operations for creating objects from four arguments.

#include <CGAL/function_objects.h>

Requirements

Result defines a corresponding constructor.

Creator_4<Arg1, Arg2, Arg3, Arg4, Result>:: argument1_type

type of first argument.

Creator_4<Arg1, Arg2, Arg3, Arg4, Result>:: argument2_type

type of second argument.

Creator_4<Arg1, Arg2, Arg3, Arg4, Result>:: argument3_type

type of third argument.

Creator_4<Arg1, Arg2, Arg3, Arg4, Result>:: argument4_type

type of 4th argument.

Creator_4<Arg1, Arg2, Arg3, Arg4, Result>:: result_type

type of object to create.

result_type *c.operator()(*
argument_type1 a1,
argument_type2 a2,
argument_type3 a3,
argument_type4 a4) const

returns *result_type(a1, a2, a3, a4)*.

CGAL::Creator_5<Arg1, Arg2, Arg3, Arg4, Arg5, Result>

Functor Class

Definition

The concept *Creator_5<Arg1, Arg2, Arg3, Arg4, Arg5, Result>* defines types and operations for creating objects from five arguments.

#include <CGAL/function_objects.h>

Requirements

Result defines a corresponding constructor.

Creator_5<Arg1, Arg2, Arg3, Arg4, Arg5, Result>:: argument1_type

type of first argument.

Creator_5<Arg1, Arg2, Arg3, Arg4, Arg5, Result>:: argument2_type

type of second argument.

Creator_5<Arg1, Arg2, Arg3, Arg4, Arg5, Result>:: argument3_type

type of third argument.

Creator_5<Arg1, Arg2, Arg3, Arg4, Arg5, Result>:: argument4_type

type of 4th argument.

Creator_5<Arg1, Arg2, Arg3, Arg4, Arg5, Result>:: argument5_type

type of 5th argument.

Creator_5<Arg1, Arg2, Arg3, Arg4, Arg5, Result>:: result_type

type of object to create.

result_type c.operator()(
argument_type1 a1,
argument_type2 a2,
argument_type3 a3,
argument_type4 a4,
argument_type5 a5) const

returns *result_type(a1, a2, a3, a4, a5)*.

Functor
Class

CGAL::Creator_uniform_2<Arg, Result>

Definition

The concept *Creator_uniform_2<Arg, Result>* defines types and operations for creating objects from two arguments of the same type.

#include <CGAL/function_objects.h>

Requirements

Result defines a constructor from two *Arg* arguments.

Creator_uniform_2<Arg, Result>:: argument_type

type of arguments; typedef to *Arg*.

Creator_uniform_2<Arg, Result>:: result_type type of object to create; typedef to *Result*.

result_type *c.operator()(argument_type a1, argument_type a2) const*

returns *result_type(a1, a2)*.

CGAL::Creator_uniform_3<Arg, Result>

Functor
Class

Definition

The concept *Creator_uniform_3<Arg, Result>* defines types and operations for creating objects from three arguments of the same type.

#include <CGAL/function_objects.h>

Requirements

Result defines a constructor from three *Arg* arguments.

Creator_uniform_3<Arg, Result>:: argument_type

type of arguments; typedef to *Arg*.

Creator_uniform_3<Arg, Result>:: result_type type of object to create; typedef to *Result*.

result_type *c.operator()(argument_type a1, argument_type a2, argument_type a3) const*

returns *result_type(a1, a2, a3)*.

Functor
Class

CGAL::Creator_uniform_4<Arg, Result>

Definition

The concept *Creator_uniform_4<Arg, Result>* defines types and operations for creating objects from four arguments of the same type.

#include <CGAL/function_objects.h>

Requirements

Result defines a constructor from four *Arg* arguments.

Creator_uniform_4<Arg, Result>:: argument_type

type of arguments; typedef to *Arg*.

Creator_uniform_4<Arg, Result>:: result_type type of object to create; typedef to *Result*.

result_type *c.operator()(argument_type a1, argument_type a2, argument_type a3, argument_type a4) const*

returns *result_type(a1, a2, a3, a4)*.

CGAL::Creator_uniform_5<Arg, Result>

Definition

The concept *Creator_uniform_5<Arg, Result>* defines types and operations for creating objects from five arguments of the same type.

#include <CGAL/function_objects.h>

Requirements

Result defines a constructor from five *Arg* arguments.

Creator_uniform_5<Arg, Result>:: argument_type

type of arguments; typedef to *Arg*.

Creator_uniform_5<Arg, Result>:: result_type type of object to create; typedef to *Result*.

result_type *c.operator()(argument_type a1, argument_type a2, argument_type a3, argument_type a4, argument_type a5) const*

returns *result_type(a1, a2, a3, a4, a5)*.

Functor
Class

CGAL::Creator_uniform_6<Arg, Result>

Definition

The concept *Creator_uniform_6<Arg, Result>* defines types and operations for creating objects from six arguments of the same type.

#include <CGAL/function_objects.h>

Requirements

Result defines a constructor from six *Arg* arguments.

Creator_uniform_6<Arg, Result>:: argument_type

type of arguments; typedef to *Arg*.

Creator_uniform_6<Arg, Result>:: result_type — type of object to create; typedef to *Result*.

result_type *c.operator()(argument_type a1, argument_type a2, argument_type a3, argument_type a4, argument_type a5, argument_type a6) const*

returns *result_type(a1, a2, a3, a4, a5, a6)*.

CGAL::Creator_uniform_7<Arg, Result>

Functor
Class

Definition

The concept *Creator_uniform_7<Arg, Result>* defines types and operations for creating objects from seven arguments of the same type.

#include <CGAL/function_objects.h>

Requirements

Result defines a constructor from seven *Arg* arguments.

Creator_uniform_7<Arg, Result>:: argument_type

type of arguments; typedef to *Arg*.

Creator_uniform_7<Arg, Result>:: result_type type of object to create; typedef to *Result*.

result_type c.operator()(argument_type a1, argument_type a2, argument_type a3, argument_type a4, argument_type a5, argument_type a6, argument_type a7) const

returns *result_type(a1, a2, a3, a4, a5, a6, a7)*.

CGAL::Creator_uniform_8<Arg, Result>

Definition

The concept *Creator_uniform_8<Arg, Result>* defines types and operations for creating objects from eight arguments of the same type.

#include <CGAL/function_objects.h>

Requirements

Result defines a constructor from eight *Arg* arguments.

Creator_uniform_8<Arg, Result>:: argument_type

type of arguments; typedef to *Arg*.

Creator_uniform_8<Arg, Result>:: result_type — type of object to create; typedef to *Result*.

result_type *c.operator()(*
argument_type a1,
argument_type a2,
argument_type a3,
argument_type a4,
argument_type a5,
argument_type a6,
argument_type a7,
argument_type a8) const

returns *result_type(a1, a2, a3, a4, a5, a6, a7, a8)*.

CGAL::Creator_uniform_9<Arg, Result>

Functor
Class

Definition

The concept *Creator_uniform_9<Arg, Result>* defines types and operations for creating objects from nine arguments of the same type.

#include <CGAL/function_objects.h>

Requirements

Result defines a constructor from nine *Arg* arguments.

Creator_uniform_9<Arg, Result>:: argument_type

type of arguments; typedef to *Arg*.

Creator_uniform_9<Arg, Result>:: result_type — type of object to create; typedef to *Result*.

result_type *c.operator()(*
argument_type a1,
argument_type a2,
argument_type a3,
argument_type a4,
argument_type a5,
argument_type a6,
argument_type a7,
argument_type a8,
argument_type a9) const

returns *result_type(a1, a2, a3, a4, a5, a6, a7, a8, a9)*.

Functor
Class

CGAL::Creator_uniform_d<Arg, Result>

Definition

The concept *Creator_uniform_d<Arg, Result>* defines types and operations for creating objects from two arguments of the same type.

#include <CGAL/function_objects.h>

Requirements

Result defines a constructor from three arguments: one *d* dimension and two *Arg* arguments.

Creator_uniform_d<Arg, Result>:: argument_type

type of arguments; typedef to *Arg*.

Creator_uniform_d<Arg, Result>:: result_type type of object to create; typedef to *Result*.

result_type *c.operator()(argument_type a1, argument_type a2) const*

returns *result_type(d, a1, a2)*.

CGAL::Twotuple<T>

This class is deprecated, and will be removed in some future CGAL release. Please use *CGAL::cpp0x::array* instead.

deprecated

Definition

The *Twotuple<T>* class stores a homogeneous (same type) pair of objects of type T. A *Twotuple<T>* is much like a container, in that it ”owns” its elements. It is not actually a model of container, though, because it does not support the standard methods (such as iterators) for accessing the elements of a container.

#include <CGAL/Twotuple.h>

Requirements

T must be *Assignable*.

Types

typedef T	*value_type;*

Variables

T	*e0;*	first element
T	*e1;*	second element

Creation

Twotuple<T> t;	introduces a *Twotuple<T>* using the default constructor of the elements.
Twotuple<T> t(T x, T y);	constructs a *Twotuple<T>* such that *e0* is constructed from *x* and *e1* is constructed from *y*.

deprecated

Class

CGAL::Threetuple<T>

This class is deprecated, and will be removed in some future CGAL release. Please use *CGAL::cpp0x::array* instead.

deprecated

Definition

The *Threetuple<T>* class stores a homogeneous (same type) triple of objects of type *T*. A *Threetuple<T>* is much like a container, in that it "owns" its elements. It is not actually a model of container, though, because it does not support the standard methods (such as iterators) for accessing the elements of a container.

#include <CGAL/Threetuple.h>

Requirements

T must be *Assignable*.

Types

typedef T *value_type;*

Variables

T	*e0;*	first element
T	*e1;*	second element
T	*e2;*	third element

Creation

Threetuple<T> t;

introduces a *Threetuple<T>* using the default constructor of the elements.

Threetuple<T> t(T x, T y, T z);

constructs a *Threetuple<T>* such that *e0* is constructed from *x*, *e1* is constructed from *y* and *e2* is constructed from *z*.

deprecated

Class

CGAL::Fourtuple<T>

This class is deprecated, and will be removed in some future CGAL release. Please use *CGAL::cpp0x::array* instead.

deprecated

Definition

The *Fourtuple<T>* class stores a homogeneous (same type) fourtuple of objects of type *T*. A *Fourtuple<T>* is much like a container, in that it "owns" its elements. It is not actually a model of container, though, because it does not support the standard methods (such as iterators) for accessing the elements of a container.

#include <CGAL/Fourtuple.h>

Requirements

T must be *Assignable*.

Types

typedef T *value_type;*

Variables

T	*e0;*	first element
T	*e1;*	second element
T	*e2;*	third element
T	*e3;*	fourth element

Creation

Fourtuple<T> t;

introduces a *Fourtuple<T>* using the default constructor of the elements.

Fourtuple<T> t(T x, T y, T z, T t);

constructs a *Fourtuple<T>* such that *e0* is constructed from *x*, *e1* from *y*, *e2* from *z* and *e3* from *t*.

deprecated

Class

CGAL::Sixtuple<T>

This class is deprecated, and will be removed in some future CGAL release. Please use *CGAL::cpp0x::array* instead.

deprecated

Definition

The *Sixtuple<T>* class stores a homogeneous (same type) sixtuple of objects of type *T*. A *Sixtuple<T>* is much like a container, in that it "owns" its elements. It is not actually a model of container, though, because it does not support the standard methods (such as iterators) for accessing the elements of a container.

#include <CGAL/Sixtuple.h>

Requirements

T must be *Assignable*.

Types

typedef T	*value_type;*

Variables

T	*e0;*	first element
T	*e1;*	second element
T	*e2;*	third element
T	*e3;*	fourth element
T	*e4;*	fifth element
T	*e5;*	sixth element

Creation

Sixtuple<T> t;

introduces a *Sixtuple<T>* using the default constructor of the elements.

Sixtuple<T> t(T x, T y, T z, T t, T u, T v);

constructs a *Sixtuple<T>* such that *e0* is constructed from *x*, *e1* from *y*, *e2* from *z*, *e3* from *t*, *e4* from *u* and *e5* from *v*.

deprecated

Class

CGAL::Triple<T1, T2, T3>

Class

Definition

The Triple class is an extension of *std::pair*. *Triple<T1, T2, T3>* is a heterogeneous triple: it holds one object of type *T1*, one of type *T2*, and one of type *T3*. A *Triple<T1, T2, T3>* is much like a container, in that it "owns" its elements. It is not actually a model of container, though, because it does not support the standard methods (such as iterators) for accessing the elements of a container.

This class is NOT RECOMMENDED anymore. We recommend that you use *CGAL::cpp0x::tuple* or *CGAL::cpp0x::array* instead for new uses.

#include <CGAL/utility.h>

Requirements

T1, *T2* and *T3* must be *Assignable*. Additional operations have additional requirements.

Types

typedef T1	*first_type;*
typedef T2	*second_type;*
typedef T3	*third_type;*

Variables

T1	*first;*	first element. Please access it using *get<0>()*.
T2	*second;*	second element. Please access it using *get<1>()*.
T3	*third;*	third element. Please access it using *get<2>()*.

Creation

Triple<T1, T2, T3> t;

introduces a triple using the default constructor of the three elements.

Triple<T1, T2, T3> t(T1 x, T2 y, T3 z);

constructs a triple such that *first* is constructed from *x*, *second* is constructed from *y*, and *third* is constructed from *z*.

template <class U, class V, class W>
Triple<T1, T2, T3> t(U u, V v, W w);

constructs a triple such that *first* is constructed from *u*, *second* is constructed from *v*, and *third* is constructed from *w*. *Requirement*: Proper conversion operators exist from *U* to *T1*, *V* to *T2*, and *W* to *T3*.

template <int i>
T *t.get()*

Gives access to *first*, *second* or *third* whenever *i* is 0, 1 or 2, via a, potentially const, reference. Note: *T* stands for *T1*, *T2* or *T3* depending on *i*.

template <class T1, class T2, class T3>
bool $x < y$

The comparison operator. It uses lexicographic comparison: the return value is true if the first element of *x* is less than the first element of *y*, and false if the first element of *y* is less than the first element of *x*. If neither of these is the case, then it returns true if the second element of *x* is less than the second element of *y*, and false if the second element of *y* is less than the second element of *x*. If neither of these is the case, then it returns the result of comparing the third elements of *x* and *y*. This operator may only be used if *T1*, *T2* and *T3* define the comparison operator.

template <class T1, class T2, class T3>
bool $x == y$

The equality operator. The return value is true if and only the first elements of *x* and *y* are equal, the second elements of *x* and *y* are equal, and the third elements of *x* and *y* are equal. This operator may only be used if *T1*, *T2* and *T3* define the equality operator.

template <class T1, class T2, class T3>
Triple<T1, T2, T3>

make_triple(T1 x, T2 y, T3 z)

Equivalent to *Triple<T1, T2, T3>(x, y, z)*.

template <class T1, class T2, class T3>
Triple<T1, T2, T3>

make_tuple(T1 x, T2 y, T3 z)

Equivalent to *Triple<T1, T2, T3>(x, y, z)*.

CGAL::Quadruple<T1, T2, T3, T4>

Class

Definition

The Quadruple class is an extension of *std::pair*. *Quadruple<T1, T2, T3, T4>* is a heterogeneous quadruple: it holds one object of type *T1*, one of type *T2*, one of type *T3*, and one of type *T4*. A *Quadruple<T1, T2, T3, T4>* is much like a container, in that it “owns” its elements. It is not actually a model of container, though, because it does not support the standard methods (such as iterators) for accessing the elements of a container.

This class is NOT RECOMMENDED anymore. We recommend that you use *CGAL::cpp0x::tuple* or *CGAL::cpp0x::array* instead for new uses.

#include <CGAL/utility.h>

Requirements

T1, *T2*, *T3* and *T4* must be *Assignable*. Additional operations have additional requirements.

Types

typedef T1	*first_type;*
typedef T2	*second_type;*
typedef T3	*third_type;*
typedef T4	*fourth_type;*

Variables

T1	*first;*	first element. Please access it using *get<0>()*.
T2	*second;*	second element. Please access it using *get<1>()*.
T3	*third;*	third element. Please access it using *get<2>()*.
T4	*fourth;*	fourth element. Please access it using *get<3>()*.

Creation

Quadruple<T1, T2, T3, T4> t;

introduces a quadruple using the default constructor of the four elements.

Quadruple<T1, T2, T3, T4> t(T1 x, T2 y, T3 z, T4 w);

constructs a quadruple such that *first* is constructed from *x*, *second* is constructed from *y*, *third* is constructed from *z*, and *fourth* is constructed from *w*.

template <class U, class V, class W, class X>
Quadruple<T1, T2, T3, T4> t(U u, V v, W w, X x);

constructs a quadruple such that *first* is constructed from *u*, *second* is constructed from *v*, *third* is constructed from *w*, and *fourth* is constructed from *x*.
Requirement: Proper conversion operators exist from *U* to *T1*, *V* to *T2*, *W* to *T3*, and *X* to *T4*.

template <int i>
T *t.get()*

Gives access to *first*, *second*, *third* or *fourth* whenever *i* is 0, 1, 2 or 3, via a, potentially const, reference. Note: *T* stands for *T1*, *T2*, *T3* or *T4* depending on *i*.

template <class T1, class T2, class T3, class T4>
bool $x < y$

The comparison operator. It uses lexicographic comparison: the return value is true if the first element of *x* is less than the first element of *y*, and false if the first element of *y* is less than the first element of *x*. If neither of these is the case, then it returns true if the second element of *x* is less than the second element of *y*, and false if the second element of *y* is less than the second element of *x*. If neither of these is the case, then it returns true if the third element of *x* is less than the third element of *y*, and false if the third element of *y* is less than the third element of *x*. If neither of these is the case, then it returns the result of comparing the fourth elements of *x* and *y*. This operator may only be used if *T1*, *T2*, *T3*, and *T4* define the comparison operator.

template <class T1, class T2, class T3, class T4>
bool $x == y$

The equality operator. The return value is true if and only the first elements of *x* and *y* are equal, the second elements of *x* and *y* are equal, the third elements of *x* and *y* are equal, and the fourth elements of *x* and *y* are equal. This operator may only be used if *T1*, *T2*, *T3*, and *T4* define the equality operator.

template <class T1, class T2, class T3, class T4>
Quadruple<T1, T2, T3, T4>

make_quadruple(T1 x, T2 y, T3 z, T4 w)

Equivalent to *Quadruple<T1, T2, T3, T4>(x, y, z, w)*.

template <class T1, class T2, class T3, class T4>
Quadruple<T1, T2, T3, T4>

make_tuple(T1 x, T2 y, T3 z, T4 w)

Equivalent to *Quadruple<T1, T2, T3, T4>(x, y, z, w)*.

CGAL::Boolean_tag<bool value>

Class

Definition

Depending on *bool value* the class *Boolean_tag<bool value>* indicates that something is *true* or *false* respectively.

#include <CGAL/tags.h>

Constants

static const bool *value;*

See Also

CGAL::Tag_true . . . . page 4232
CGAL::Tag_false . . . . page 4233

Class

CGAL::Tag_true

Definition

The typedef *Tag_true* is *Boolean_tag<true>*. It is used to indicate, for example, that a certain feature is available in a class.

#include <CGAL/tags.h>

static const bool	*value;*	is *true*

See Also

CGAL::Boolean_tag<bool value> .... page 4231
CGAL::Tag_false .... page 4233

CGAL::Tag_false

Class

Definition

The typedef *Tag_false* is *Boolean_tag<false>*. It is used to indicate, for example, that a certain feature is not available in a class.

#include <CGAL/tags.h>

static const bool	*value;*	is *false*

See Also

CGAL::Boolean_tag<bool value> . . . . page 4231
CGAL::Tag_true . . . . page 4232

Class

CGAL::Null_functor

Definition

Class indicating the absence of a functor.

Is Model for the Concepts

DefaultConstructible

See Also

AlgebraicStructureTraits . . . . page 78
RealEmbeddableTraits . . . . page 124

CGAL::Null_tag

Class

Definition

General tag indicating that non of any other possible tags is valid.

Is Model for the Concepts

DefaultConstructible

See Also

AlgebraicStructureTraits . . . . page 78

Class

CGAL::Uncertain<T>

Definition

An object of the class *Uncertain<T>* represents an uncertainty on the value of type *T*. This uncertainty is represented by a non-empty range of values of type *T*.

The idea is that sometimes you are not sure of the result of a function, and you would like to communicate that to the caller. *Uncertain<T>* allows just that.

Uncertain<T> is also meant to be used as a drop-in replacement for *T* in some template contexts, as much as possible. This is why it provides overloaded operators and functions to naturally extend the Boolean operations for *Uncertain<bool>* for example, or the operations on enumeration types.

Uncertain<T> is used in CGAL as the return type of geometric predicates when the number type used is interval arithmetic like *Interval_nt*. End users typically do not see it, as it is hidden in the implementation of the filtered predicates provided by the various filtered kernels, but it is important that providers of predicates that are meant to be filtered by *Filtered_predicate*, know about it.

Note concerning CGAL assertions: assertions checking an expression of type *Uncertain<bool>* will trigger an assertion failure only if the assertion is certainly *false*. In case of an indeterminate value, the assertion is not triggered. This means that we assume, in case of doubt, that there is no error.

It can also be used in other contexts as well, as it is a general tool. This can be seen as support for non-deterministic programming. Finally, note that this class has some common points with *boost::tribool*.

#include <CGAL/Uncertain.h>

Parameters

The parameter *T* can either be *bool* or one of the three-valued (-1, 0, 1) enumeration types: *Sign*, *Comparison_result*, *Orientation*, *Oriented_side*, *Bounded_side* or *Angle*.

Some functions are defined only when *T* is *bool* or alternatively when it is one of the enumeration types listed previously.

Types

Uncertain<T>:: value_type — The type *T*.

Uncertain<T>:: Uncertain_conversion_exception

The type of the exception thrown for uncertain conversions. It is a typedef to the type *CGAL::Uncertain_conversion_exception* which derives from *std::range_error*.

Creation

Uncertain<T> u; — introduces a certain object with value *T()*.

Uncertain<T> u(T t); introduces a certain object with value t.

Uncertain& $u = T\, t$ assigns the certain value t to u.

Uncertain<T> u(T i, T s); introduces an object representing the range with lower bound i and upper bound s.
Precondition: $i <= s$.

Access Functions

The following functions are meant to be used very rarely, they provide ways to inspect the content of an *Uncertain<T>* object.

T	*u.inf() const*	returns the lower bound of the range represented by u.
T	*u.sup() const*	returns the upper bound of the range represented by u.
bool	*u.is_same(Uncertain u) const*	

returns true whether u and u are the same range (equality as sets).

Uncertainty testing and conversion

There are several ways to extract the content of an *Uncertain* object. The simplest way is to rely on the implicit conversion from *Uncertain<T>* to *T*. In this case, no special code has to be written, apart from an exception handler (anywhere higher in the call stack) to manage the uncertain case. The more involved one is more efficient, but requires manual treatment of the uncertain case, such as:

```
Uncertain<bool> b = ...;
if (is_certain(b))
  bool cert_b = get_certain(b); // Extract the certain bool it contains
  ...
else
  ... // b is indeterminate
```

Another option is :

```
Uncertain<bool> b = ...;
if (certainly(b))
  ... // b is certainly true
else if (certainly_not(b))
  ... // b is certainly false
else
  ... // b is indeterminate
```

There are many other handy functions which can be used for easier usage depending on the context. They are listed in the sequel.

bool *u.is_certain() const* returns *true* iff the value is certain, that is, it is unique, the range is a singleton. That is, *u.inf() == u.sup()*.

T	*u.make_certain() const*	if *u.is_certain()*, then returns the certain value which is represented. Otherwise, throws an exception of type *Uncertain_conversion_exception*. A profile counter of the number of such exceptions thrown during the execution of the program is available with *CGAL_PROFILE*.
T	*T(u)*	conversion operator to *T*. It does and returns the same thing as *u.make_certain()*. Note that relying on the automatic conversion can throw exceptions, which defeats the purpose of propagating uncertainty. Nevertheless, in many cases, it is hard to avoid it, for example for the `&&` and || operators for *bool* (see below).

Static member function

static Uncertain<T> *Uncertain<T>::indeterminate()*

returns an indeterminate range.

Free functions

template <class T> *T*	*inf(Uncertain<T> u)*	returns *u.inf()*.
template <class T> *T*	*sup(Uncertain<T> u)*	returns *u.sup()*.
template <class T> *bool*	*is_certain(T t)*	returns *true*.
template <class T> *bool*	*is_certain(Uncertain<T> u)*	returns *u.is_certain*().
template <class U> *U*	*indeterminate()*	returns *U::indeterminate()* if *U* is *Uncertain<T>*, and *U()* otherwise.
template <class T> *bool*	*is_indeterminate(T u)*	returns *false*.
template <class T> *bool*	*is_indeterminate(Uncertain<T> u)*	returns *!is_certain(u)*.

template <class T>

T *get_certain(T t)* returns *t*.

template <class T>
T *get_certain(Uncertain<T> u)*

returns *u.make_certain*().
Precondition: *u.is_certain*().

template <class T>
T *make_certain(T t)* returns *t*.

template <class T>
T *make_certain(Uncertain<T> u)*

returns *u.make_certain*().

template <class T>
Uncertain<T> *make_uncertain(T t)* returns *Uncertain<T>(u)*.

template <class T>
Uncertain<T> *make_uncertain(Uncertain<T> u)*

returns *u*.

Overloaded operators

The overloaded operators and functions are defined as preserving the set-inclusion property. Similarly to interval arithmetic, the returned range is guaranteed to contain the result of the operation over all values of the input range(s). In the following documentation we express this as the extension of the corresponding function over the type *T*.

template <class T>
Uncertain<bool> *Uncertain<T> u == Uncertain<T> v*

returns the extension of the equality operator over *u* and *v*.

template <class T>
Uncertain<bool> *Uncertain<T> u == T v* returns *u* == *make_uncertain(v)*.

template <class T>
Uncertain<bool> *T u == Uncertain<T> v* returns *v* == *u*.

template <class T>
Uncertain<bool> *Uncertain<T> u != Uncertain<T> v*

returns the extension of the inequality operator over *u* and *v*.

template <class T>

Uncertain<bool> *Uncertain<T> u* $!=$ *T v* returns *u != make_uncertain(v)*.

template <class T>
Uncertain<bool> *T u* $!=$ *Uncertain<T> v* returns *v != u*.

Overloaded operators for *Uncertain<bool>* only

Uncertain<bool> ! *Uncertain<bool> u* returns the range containing the negated values of *u*.

Uncertain<bool> *Uncertain<bool> u* | *Uncertain<bool> v*

returns the range containing the values computed as logical or from *u* and *v*.

Uncertain<bool> *Uncertain<bool> u* | *bool v*

returns *u* — *make_uncertain(v)*.

Uncertain<bool> *bool u* | *Uncertain<bool> v*

returns *v* — *u*.

Uncertain<bool> *Uncertain<bool> u* & *Uncertain<bool> v*

returns the range containing the values computed as logical and from *u* and *v*.

Uncertain<bool> *Uncertain<bool> u* & *bool v*

returns *u* & *make_uncertain(v)*.

Uncertain<bool> *bool u* & *Uncertain<bool> v*

returns *v* & *u*.

Note : the logical operators && and || are not overloaded on purpose. The reason is that, when *f()* && *g()* is evaluated and they return *bool*, then *g()* is only evaluated when *f()* returns *true*. One could have a dependency so that *g()* has an internal precondition that required that *f()* had returned *true*. The overloaded operators for user-defined types can not provide this short-circuiting property, and so, if the overloaded operators where provided, then *g()* would be evaluated, no matter the result of *f()*, which could lead to an unwanted situation, or a performance loss. The & and | operators do not have this short-circuiting property, and are therefore overloaded safely.

When translating normal code to use and propagate uncertainty, such as :

```
// Logical AND
if ( (p.x() == 0) && (p.y() == 0) )
  ...
else
  ...

// Logical OR
if ( (q.x() == 0) || (q.y() == 0) )
  ...
else
  ...
```

One can do, for example :

```
// Logical AND
Uncertain<bool> tmp = (p.x() == 0);
Uncertain<bool> res = certainly_not(tmp) ? make_uncertain(false) : tmp
 & (p.y() == 0);

... // Use res

// Logical OR
Uncertain<bool> tmp = (q.x() == 0);
Uncertain<bool> res = certainly(tmp) ? make_uncertain(true) : tmp |
 (q.y() == 0);

... // Use res
```

This ensures that the first expression is not evaluated twice, and that the second is evaluated only if needed.

This behavior can also be emulated through the use of macros, but only using non-standard features ("statement expressions", such as provided by GCC). The macros *CGAL_AND* and *CGAL_OR* are provided that perform the lazy evaluation of these logical operations. On compilers that do not support statement expressions, the macros simply expand to the && and || operators (which will throw an exception instead of propagating the uncertainty).

```
// Logical AND
Uncertain<bool> res = CGAL_AND( p.x() == 0 , p.y() == 0 );

... // Use res

// Logical OR
Uncertain<bool> res = CGAL_OR( q.x() == 0 , q.y() == 0 );

... // Use res
```

For convenience, the macros *CGAL_AND_3* and *CGAL_OR_3* are also provided to support boolean operations with 3 arguments instead of 2.

Overloaded operators and functions for *Uncertain<enum T>* **only**

template <class T>

Uncertain<bool> *Uncertain<T> u* < *Uncertain<T> v*

returns the extension of the less-than operator over *u* and *v*.

template <class T>
Uncertain<bool> *Uncertain<T> u* < *T v* returns *u* < *make_uncertain(v)*.

template <class T>
Uncertain<bool> *T u* < *Uncertain<T> v* returns *make_uncertain(u)* < *v*.

template <class T>
Uncertain<bool> *Uncertain<T> u* > *Uncertain<T> v*

returns the extension of the greater-than operator over *u* and *v*.

template <class T>
Uncertain<bool> *Uncertain<T> u* > *T v* returns *u* > *make_uncertain(v)*.

template <class T>
Uncertain<bool> *T u* > *Uncertain<T> v* returns *make_uncertain(u)* > *v*.

template <class T>
Uncertain<bool> *Uncertain<T> u* <= *Uncertain<T> v*

returns the extension of the less-than or equal operator over *u* and *v*.

template <class T>
Uncertain<bool> *Uncertain<T> u* <= *T v* returns *u* <= *make_uncertain(v)*.

template <class T>
Uncertain<bool> *T u* <= *Uncertain<T> v* returns *make_uncertain(u)* <= *v*.

template <class T>
Uncertain<bool> *Uncertain<T> u* >= *Uncertain<T> v*

returns the extension of the greater-than or equal operator over *u* and *v*.

template <class T>
Uncertain<bool> *Uncertain<T> u* >= *T v* returns *u* > *make_uncertain(v)*.

template <class T>
Uncertain<bool> *T u* >= *Uncertain<T> v* returns *make_uncertain(u)* >= *v*.

template <class T>

Uncertain<T> *Uncertain<T> u* $*$ *Uncertain<T> v*

returns the extension of the multiplication operator over *u* and *v*. This requires *T* to have a multiplication operator as well.

template <class T>
Uncertain<T> *Uncertain<T> u* $*$ *T v* returns *u* * *make_uncertain(v)*.

template <class T>
Uncertain<T> *T u* $<$ *Uncertain<T> v* returns *make_uncertain(u)* * *v*.

template <class T>
Uncertain<T> $-$*Uncertain<T> u* returns the extension of the unary minus operator over *u*.

template <class T, class U>
Uncertain<T> *enum_cast(Uncertain<U> u)*

returns the extension of the *enum_cast<T>* function over *u*.

Other free functions for *Uncertain<bool>*

bool *certainly(Uncertain<bool> u)*

returns *true* iff *u.is_certain()*, and the *u.make_certain*() returns *true*.

bool *certainly(bool u)* returns *u*.

bool *possibly(Uncertain<bool> u)*

returns *true* iff *u.is_certain()* returns *false*, or if *u.make_certain*() returns *true*.

bool *possibly(bool u)* returns *u*.

bool *certainly_not(Uncertain<bool> u)*

returns *true* iff *u.is_certain()*, and the *u.make_certain*() returns *false*.

bool *certainly_not(bool u)* returns *!u*.

bool *possibly_not(Uncertain<bool> u)*

returns *true* iff *u.is_certain()* returns *false*, or if *u.make_certain*() returns *false*.

bool *possibly_not(bool u)* returns *!u*.

See Also

CGAL::Interval_nt<bool>

CGAL::cpp11::array<T, int>

Class

Definition

An object of the class *cpp11::array<T, int>* represents an array of elements of type *T*, the number of which is specified by the second template argument.

There is actually no class in namespace *CGAL::cpp11* with this name, but a using declaration which imports a class from another namespace. By order of priority: the one in namespace *std* is used (provided by C++0x), if not found, then the one in namespace *std::tr1* is used (provided by TR1), and finally, the fallback solution is taken from Boost.

#include <CGAL/array.h>

Parameters

The parameter *T* is the value type. The second parameter is the dimension of the array.

Construction functions

The array class does not provide a constructor which can be used to initialize data members. CGAL therefore provides a *make_array* function for this purpose, up to a certain number of arguments.

template <class T>
array<T, 1> *make_array(T a)* returns an array of dimension 1 whose first element is *a*.

template <class T>
array<T, 2> *make_array(T a1, T a2)*

returns an array of dimension 2 whose first element is *a1* and second element is *a2*.

Class

CGAL::cpp11::tuple<...>

Definition

An object of the class *cpp11::tuple<...>* represents a heterogeneous tuple of elements of the types specified in parameters, which are in variadic number.

There is actually no class in namespace *CGAL::cpp11* with this name, but a using declaration which imports a class from another namespace. By order of priority: the one in namespace *std* is used (provided by C++0x), if not found, then the one in namespace *std::tr1* is used (provided by TR1), and finally, the fallback solution is taken from Boost.

#include <CGAL/tuple.h>

Parameters

The parameters ... are the value types.

Free functions and helper classes

Some free functions part of the standard interface of *tuple* are also brought in namespace *CGAL::cpp11* with using declarations, these are *make_tuple*, *get*, *tie*. Like in C++0x, the *get* function template is specialized so that it can take *std::pair* as argument. Two standard helper classes are also provided for convenience (*tuple_size* and *tuple_element*).

CGAL::Fast

Definition

Fast is a tag class. It can be used to parameterize a complexity policy in order to specify a particularly fast variant of an algorithm. For example, passing *Location_policy<Fast>* as parameter to *Delaunay_triangulation_3* selects a fast point location at the expense of more memory usage.

#include <CGAL/Complexity_tags.h>

Is Model for the Concepts

DefaultConstructible, CopyConstructible

See Also

Location_policy
Compact
Compact_location
Fast_location

Class

CGAL::Compact

Definition

Compact is a tag class. It can be used to parameterize a complexity policy in order to specify a particularly memory compact variant of an algorithm. For example, passing *Location_policy<Compact>* as parameter to *Delaunay_triangulation_3* selects a slower point location which saves memory.

#include <CGAL/Complexity_tags.h>

Is Model for the Concepts

DefaultConstructible, CopyConstructible

See Also

Location_policy
Fast
Fast_location
Compact_location

CGAL::Location_policy<Tag>

Class

Definition

Location_policy<Tag> is a policy class which can be used to specify a trade-off between memory usage and time complexity for the point location strategy used in a data-structure. For example, passing *Location_policy< Compact>* as parameter to *Delaunay_triangulation_3* selects a slower point location which saves memory.

#include <CGAL/Location_policy.h>

Parameters

Tag can only be either *Fast* or *Compact* currently.

Is Model for the Concepts

DefaultConstructible, CopyConstructible

See Also

Compact
Fast
Fast_location
Compact_location

Class

CGAL::Fast_location

Definition

A typedef to *Location_policy<Fast>*.

#include <CGAL/Location_policy.h>

See Also

Compact
Fast
Location_policy
Compact_location

CGAL::Compact_location

Class

Definition

A typedef to *Location_policy<Compact>*.

#include <CGAL/Location_policy.h>

See Also

Compact
Fast
Location_policy
Fast_location

Class

CGAL::Default

Definition

Default is a tag class. It can be used to state that one wants to use the default argument of a template parameter of a class template.

This can be useful in several cases: (a) when one needs a non-default value for another template parameter coming next (since C++ only supports defaults at the end of lists), (b) when the default is actually a complex expression, e.g. refering to previous template parameters (in this case, it shortens compiler error messages and mangled symbol names), (c) when defining the default involves circular dependencies of type instantiations (there, it breaks the cycle in a nice way).

Using the mechanism is easy : just plug *Default* as template argument in the place where you would like to use the default. You should refer to the documentation of the template class you are using in order to know whether this functionality is offered.

Also beware that the type of the instantiated template class will not be the same when instantiating it using *Default* instead of the type of the default argument, even though their interfaces will otherwise be the same. This may have consequences in some cases.

Is Model for the Concepts

DefaultConstructible, CopyConstructible

advanced

In order to help the template class writer, *Default* provides a convenient way to extract the desired type for a template parameter which may be defaulted using *Default*. It is enough to fetch the type as *Default::Get< Parameter, Type>::type*, as in the example program below.

Types

Default:: template <typename Parameter, typename Type> struct Get;

A nested template providing a typedef *type* which equals *Type* if *Parameter* is *Default*, and *Parameter* otherwise.

advanced

Example

```
#include <CGAL/Default.h>

// A is a concrete type
struct A {};

// B is the template class which has 2 template parameters
// with default arguments : A and int.
```

```
template < typename A1_ = A, typename A2 = int >
struct B
{
    B()
      : a1()
    {}

    // Note that it is also possible to use CGAL::Default
    // instead of A as the default argument for A1_ above.

    // Extract the desired type for A1 :
    typedef typename CGAL::Default::Get<A1_, A>::type  A1;

    A1 a1;
};

int main ()
{
    B<CGAL::Default, double> b;

    A a = b.a1; //  It is really of type A.
}
```

File: examples/STL_Extension/Default.cpp

Chapter 74

CGAL and the Boost Graph Library

Andreas Fabri and Fernando Cacciola and Ron Wein

Contents

Many geometric data structures can be interpreted as graphs, as they consist of vertices, edges and faces. This is the case for the halfedge data structure, for the polyhedron, for arrangements and for triangulations. With means of duality one can also interpret faces as vertices and edges between adjacent faces as edges of the dual graph.

As the scope of CGAL is geometry and not graph algorithms, we provide the necessary classes and functions that allow to use the algorithms of the Boost Graph Library (BGL) [SLL02] for CGAL data structures.

74.1 A Short Introduction to the Boost Graph Library

The algorithms of the BGL operate on models of the various *graph concepts*. The *traits class boost::graph_traits* allows the algorithms to determine the types of vertices and edges. *Free functions* that operator on graphs allow the algorithms to obtain, for example, the source vertex of an edge, or all edges incident to a vertex. The algorithms use *property maps* to associate information to vertices and edges. The algorithms allow *visitors* to register callbacks that will be called during the execution of the algorithms. Finally, the graph algorithms use the *named parameter* mechanism, which allows to pass the arguments in arbitrary order.

Graph Concepts

The BGL introduces several graph concepts, which have different sets of characteristics and requirements, as for example whether one can enumerate all vertices or all edges, whether one only can get the outgoing edges of a vertex, or also the ingoing edges, or whether one can add and remove vertices and edges or not.

Graph concepts in the BGL manual: `http://www.boost.org/libs/graph/doc/graph_concepts.html`

The Graph Traits Class

The algorithms determine types with the help of the traits class *boost::graph_traits*. Such types are the *vertex_descriptor* which is equivalent to a vertex handle in CGAL data structures, the *vertex_iterator* which is similar to the vertex iterators in CGAL data structures, and the *out_edge_iterator* which is similar to edge circulators, which allow to enumerate the edges incident to a vertex. The latter two are similar and not equivalent, because their value type is a *vertex_descriptor*, whereas in CGAL handles, iterators, and cicrulators all have the same value type, namely the vertex type. Given a graph type *G* the declaration of a vertex descriptor looks as follows: *boost::graph_traits<G>::vertex_descriptor vd;*.

The graph traits in the BGL manual: `http://www.boost.org/libs/graph/doc/graph_traits.html`

Free Functions for Exploring a Graph

The algorithms obtain incidence information with the help of global functions like *pair<vertex_iterator,vertex_iterator> vertices(const Graph& g);* for getting an iterator range which allows to enumerate all vertices, or *int num_vertices(const Graph&);* for getting the number of vertices of a graph, or *vertex_descriptor source(edge_descriptor, const Graph&*, for getting the source vertex of an edge. Note, that the way we have written the types is a simplification, that is in reality the signature of the first of the above functions is *pair<boost::graph_traits<Graph>::vertex_iterator,boost::graph_traits<Graph>::vertex_iterator> vertices(const Graph& g);*.

The free functions required for graph concepts: `http://www.boost.org/libs/graph/doc/graph_concepts.html`

Property Maps

Another feature used heavily in the BGL is the *property map* which is offered by the Boost Property Map Library. Property maps are used to attach information to vertices and edges. It is again a traits class and some free functions for obtaining the property map from a graph, and for getting and putting properties.

The free functions are *get* and *put*. The first one is overloaded. One version allows to obtain a property map for a given property tag. For example *m = get(g, boost::vertex_index)* gives us a property map that associates an index in the range *[0, num_vertices(g))* to each vertex descriptor of the graph. The second version of the *get* function allows to read it as follows for a vertex descriptor *vd*: *int vdi = get(m, vd)*. Just as *get* allows to read data, *put* allows to write them. For example, the Dijksta's shortest path algorithm writes the predecessor of each vertex, as well as the distance to the source in such a property map.

The data themselves may be stored in the vertex or edge, or they may be stored in an external data structure, or they may be computed on the fly. This is an “implementation detail” of the particular property map.

Property maps in the Boost manuals: `http://www.boost.org/libs/property_map/doc/property_map.html`

Visitors

Visitors are ojects that provide functions that get called at specified event points by the algorithm they visit. The notion of visitors is a design pattern, and also used in CGAL, e.g., the *Arr_observer<Arrangement>* in the arrangement package.

The functions as well as the event points are library specific. Event points in graph algorithms are, for example, when a vertex is traversed the first time, or when all outgoing edges of a vertex are traversed.

Visitors in the BGL manual: `http://www.boost.org/libs/graph/doc/visitor_concepts.html`

Named Parameters

The algorithms of the BGL often have many parameters. Although the default value is most often appropriate, one has to write them explicitly, if one only wants to deviate from the default for the last one. The solution to this problem is to first write a tag and then the parameter, which for Dijkstra's shortest path algorithm, might look as follows:

```
std::vector<vertex_descriptor> p(num_vertices(g));
std::vector<int> d(num_vertices(g));
vertex_descriptor s = vertex(A, g);
dijkstra_shortest_paths(g, s, predecessor_map(&p[0]).distance_map(&d[0]));
```

The named parameters in the example use the tags *predecessor_map* and *distance_map* and they are concatenated with the dot operator.

Named parameters in the BGL manual: `http://www.boost.org/libs/graph/doc/bgl_named_params.html`

74.2 Extensions of the BGL

CGAL provides the partial specializations and free functions such that several data structures become model of some of the BGL graph concepts. Furthermore, we define the new graph concept *HalfedgeGraph*, a traits class *halfedge_graph_traits*, and free functions for accessing opposite edges as well as the clockwise and counterclockwise neighbor of an edge around a given vertex.

These extensions are used by the surface simplification algorithms which follow the design of the BGL as sketched in the previous section.

74.3 Header Files, Namespaces, and Naming Conventions

As we interface two libraries we have to explain what resides in which namespace, and what naming conventions we apply to what.

Partial specializations of the *boost::graph_traits<Graph>* for the CGAL package *Package* are in the namespace *boost* and in the headerfile *<CGAL/boost/graph/graph_traits_Package.h>*.

The *halfedge_graph_traits* class is in the namespace *CGAL*, but it is not capitalized as the *boost::graph_traits* is not. The same holds for the types and enums for vertex and edge properties.

74.4 Polyhedral Surfaces as Model of the Boost Graph Concept

The class *Polyhedron_3* is model of the graph concept. Furthermore this chapter introduces a new graph concept, the *HalfedgeGraph*.

74.4.1 Example: Minimum Spanning Tree of a Polyhedral Surface

The example code computes the minimum spanning tree on a polyhedral surface. More examples can be found in Chapter 53 on surface mesh simplification.

```
#include <CGAL/Cartesian.h>
#include <CGAL/Polyhedron_3.h>
#include <CGAL/IO/Polyhedron_iostream.h>
#include <CGAL/boost/graph/graph_traits_Polyhedron_3.h>
#include <CGAL/boost/graph/properties_Polyhedron_3.h>

#include <iostream>
#include <list>

#include <boost/graph/kruskal_min_spanning_tree.hpp>


typedef CGAL::Cartesian<double>                                Kernel;
typedef Kernel::Vector_3                                       Vector;
typedef Kernel::Point_3                                        Point;
typedef CGAL::Polyhedron_3<Kernel>                             Polyhedron;

typedef boost::graph_traits<Polyhedron>::vertex_descriptor
   vertex_descriptor;
typedef boost::graph_traits<Polyhedron>::vertex_iterator
   vertex_iterator;
typedef boost::graph_traits<Polyhedron>::edge_descriptor
   edge_descriptor;

// The BGL makes heavy use of indices associated to the vertices
// We use a std::map to store the index

typedef std::map<vertex_descriptor,int> VertexIndexMap;
VertexIndexMap vertex_id_map;

// A std::map is not a property map, because it is not lightweight
typedef boost::associative_property_map<VertexIndexMap>
   VertexIdPropertyMap;
VertexIdPropertyMap vertex_index_pmap(vertex_id_map);

void
kruskal(const Polyhedron& P)
{
```

```
// associate indices to the vertices
{
  vertex_iterator vb, ve;
  int index = 0;

  // boost::tie assigns the first and second element of the std::pair
  // returned by boost::vertices to the variables vit and ve
  for(boost::tie(vb,ve)=boost::vertices(P); vb!=ve; ++vb ){
    vertex_descriptor  vd = *vb;
    vertex_id_map[vd]= index++;
  }
}

// We use the default edge weight which is the squared length of the
 edge
// This property map is defined in graph_traits_Polyhedron_3.h

// In the function call you can see a named parameter: vertex_index_map
 std::list<edge_descriptor> mst;

 boost::kruskal_minimum_spanning_tree(P,
                                              std::back_inserter(mst),

  boost::vertex_index_map(vertex_index_pmap) );

 std::cout << "#VRML V2.0 utf8\n"
    "Shape {\n"
    "appearance Appearance {\n"
    "material Material { emissiveColor 1 0 0}}\n"
    "geometry\n"
    "IndexedLineSet {\n"
    "coord Coordinate {\n"
    "point [ \n";

 vertex_iterator vb, ve;
  for(boost::tie(vb,ve) = boost::vertices(P); vb!=ve; ++vb){
    std::cout << (*vb)->point() << "\n";
  }

  std::cout << "]\n"
    "}\n"
    "coordIndex [\n";

  for(std::list<edge_descriptor>::iterator it = mst.begin(); it !=
 mst.end(); ++it)
  {
    edge_descriptor e = *it ;
    vertex_descriptor s = boost::source(e,P);
    vertex_descriptor t = boost::target(e,P);
    std::cout << vertex_id_map[s] << ", " << vertex_id_map[t] <<  ",
 -1\n";
  }

  std::cout << "]\n"
```

```
        "}#IndexedLineSet\n"
        "}# Shape\n";
}


int main() {

    Polyhedron P;
    Point a(1,0,0);
    Point b(0,1,0);
    Point c(0,0,1);
    Point d(0,0,0);

    P.make_tetrahedron(a,b,c,d);

    kruskal(P);

    return 0;
}
```

File: examples/BGL_polyhedron_3/kruskal.cpp

74.4.2 Example: Using Vertices, and Edges with an ID

The following example program shows a call to the BGL Kruskal's minimum spanning tree algorithm accessing the id() field stored in a Polyhedron vertex.
The main function illustrates the access to the id() field.

```
#include <CGAL/Cartesian.h>
#include <CGAL/Polyhedron_3.h>
#include <CGAL/Polyhedron_items_with_id_3.h>
#include <CGAL/IO/Polyhedron_iostream.h>
#include <CGAL/boost/graph/graph_traits_Polyhedron_3.h>
#include <CGAL/boost/graph/properties_Polyhedron_3.h>

#include <iostream>
#include <list>

#include <boost/graph/kruskal_min_spanning_tree.hpp>

typedef CGAL::Cartesian<double>
   Kernel;
typedef Kernel::Point_3
   Point;
typedef CGAL::Polyhedron_3<Kernel,CGAL::Polyhedron_items_with_id_3>
   Polyhedron;

typedef boost::graph_traits<Polyhedron>::vertex_descriptor
   vertex_descriptor;
typedef boost::graph_traits<Polyhedron>::vertex_iterator
   vertex_iterator;
```

```
typedef boost::graph_traits<Polyhedron>::edge_descriptor
    edge_descriptor;



void
kruskal( const Polyhedron& P)
{

  // We use the default edge weight which is the squared length of the
    edge
  // This property map is defined in graph_traits_Polyhedron_3.h

  // This function call requires a vertex_index_map named parameter which
  // when  ommitted defaults to "get(vertex_index,graph)".
  // That default works here because the vertex type supports the "id()"
  // field which is used by the vertex_index internal property.
  std::list<edge_descriptor> mst;
  boost::kruskal_minimum_spanning_tree(P,std::back_inserter(mst));

  std::cout << "#VRML V2.0 utf8\n"
    "Shape {\n"
    "appearance Appearance {\n"
    "material Material { emissiveColor 1 0 0}}\n"
    "geometry\n"
    "IndexedLineSet {\n"
    "coord Coordinate {\n"
    "point [ \n";

  vertex_iterator vb, ve;
  for(boost::tie(vb,ve) = boost::vertices(P); vb!=ve; ++vb){
    std::cout << (*vb)->point() << "\n";
  }

  std::cout << "]\n"
    "}\n"
    "coordIndex [\n";

  for(std::list<edge_descriptor>::iterator it = mst.begin(); it !=
    mst.end(); ++it){
    std::cout << boost::source(*it,P)->id()
              << ", " << boost::target(*it,P)->id() <<  ", -1\n";
  }

  std::cout << "]\n"
    "}#IndexedLineSet\n"
    "}# Shape\n";
}


int main() {

  Polyhedron P;
```

```
  Point a(1,0,0);
  Point b(0,1,0);
  Point c(0,0,1);
  Point d(0,0,0);

  P.make_tetrahedron(a,b,c,d);

  // associate indices to the vertices using the "id()" field of the
     vertex.
  vertex_iterator vb, ve;
  int index = 0;

  // boost::tie assigns the first and second element of the std::pair
  // returned by boost::vertices to the variables vit and ve
  for(boost::tie(vb,ve)=boost::vertices(P); vb!=ve; ++vb ){
    vertex_descriptor  vd = *vb;
    vd->id() = index++;
  }

  kruskal(P);
  return 0;
}
```

File: examples/BGL_polyhedron_3/kruskal_with_stored_id.cpp

74.5 Triangulations as Models of the Boost Graph Concept

Triangulations have vertices and faces. Edges are pairs of a face and the index of the edge. Particular care has to be taken with the infinite vertex, and its incident edges. One can either use a *boost::filtered_graph*, which makes the infinite edges invisible, or one can have a property map that returns an infinite length for these edges.

A classical example for an algorithm that is a combination of computational geometry and graph theory is the *Euclidean Minimum Spanning Tree* for a point set in the plane. It can be computed by running the minimum spanning tree algorithm on a Delaunay triangulation of the point set.

74.5.1 Example: Euclidean Minimum Spanning Tree

In the following example we create a Delaunay triangulation and run Kruskal's minimum spanning tree algorithm on it. Because the vertex handles of the triangulation are not indices in an array, we have to provide a property map that maps vertex handles to int's in the range *[0, t.number_of_vertices())*.

```
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/Delaunay_triangulation_2.h>
#include <CGAL/boost/graph/graph_traits_Delaunay_triangulation_2.h>

#include <boost/graph/kruskal_min_spanning_tree.hpp>
#include <boost/graph/filtered_graph.hpp>

typedef CGAL::Exact_predicates_inexact_constructions_kernel K;
```

```
typedef K::Point_2 Point;

typedef CGAL::Delaunay_triangulation_2<K> Triangulation;

// As we only consider finite vertices and edges
// we need the following filter

template <typename T>
struct Is_finite {

  const T* t_;

  Is_finite()
    : t_(NULL)
  {}

  Is_finite(const T& t)
    : t_(&t)
  { }

  template <typename VertexOrEdge>
  bool operator()(const VertexOrEdge& voe) const {
    return ! t_->is_infinite(voe);
  }
};

typedef Is_finite<Triangulation> Filter;
typedef boost::filtered_graph<Triangulation,Filter,Filter>
    Finite_triangulation;
typedef boost::graph_traits<Finite_triangulation>::vertex_descriptor
    vertex_descriptor;
typedef boost::graph_traits<Finite_triangulation>::vertex_iterator
    vertex_iterator;
typedef boost::graph_traits<Finite_triangulation>::edge_descriptor
    edge_descriptor;

// The BGL makes use of indices associated to the vertices
// We use a std::map to store the index
typedef std::map<vertex_descriptor,int> VertexIndexMap;
VertexIndexMap vertex_id_map;

// A std::map is not a property map, because it is not lightweight
typedef boost::associative_property_map<VertexIndexMap>
    VertexIdPropertyMap;
VertexIdPropertyMap vertex_index_pmap(vertex_id_map);

int
main(int,char*[])
{
  Triangulation t;
  Filter is_finite(t);
  Finite_triangulation ft(t, is_finite, is_finite);

  Point p ;
```

```
  while(std::cin >> p){
    t.insert(p);
  }

  vertex_iterator vit, ve;
  // Associate indices to the vertices
  int index = 0;
  // boost::tie assigns the first and second element of the std::pair
  // returned by boost::vertices to the variables vit and ve
  for(boost::tie(vit,ve)=boost::vertices(ft); vit!=ve; ++vit ){
    vertex_descriptor  vd = *vit;
    vertex_id_map[vd]= index++;
    }


  // We use the default edge weight which is the squared length of the
     edge
  // This property map is defined in graph_traits_Triangulation_2.h

  // In the function call you can see a named parameter: vertex_index_map
   std::list<edge_descriptor> mst;
   boost::kruskal_minimum_spanning_tree(t,
                                         std::back_inserter(mst),

   vertex_index_map(vertex_index_pmap));


   std::cout << "The edges of the Euclidean mimimum spanning tree:" <<
   std::endl;

   for(std::list<edge_descriptor>::iterator it = mst.begin(); it !=
   mst.end(); ++it){
     edge_descriptor ed = *it;
     vertex_descriptor svd = boost::source(ed,t);
     vertex_descriptor tvd = boost::target(ed,t);
     Triangulation::Vertex_handle sv = svd;
     Triangulation::Vertex_handle tv = tvd;
     std::cout << "[ " << sv->point() << "  |  " << tv->point() << " ] "
   << std::endl;
   }

   return 0;
}
```

File: examples/BGL_triangulation_2/emst.cpp

74.5.2 Example: Storing the Vertex ID in the Vertex

The algorithms of the BGL extensively use of the indices of vertices. In the previous example we stored the index in a *std::map* and turned that map in a property map. This property map was then passed as argument to the shortest path function.

If the user does not pass explicitly a property map, the graph algorithms use the property map returned by the call *boost::get(boost::vertex_index,ft)*. This property map assumes that the vertex has a member function *id()* that returns a reference to an int. Therefore CGAL offers a class *Triangulation_vertex_base_with_id_2*. It is in the users responsibility to set the indices properly.

The example further illustrates that the graph traits also works for the Delaunay triangulation.

```
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/Delaunay_triangulation_2.h>
#include <CGAL/Triangulation_vertex_base_with_id_2.h>
#include <CGAL/boost/graph/graph_traits_Delaunay_triangulation_2.h>

#include <climits>
#include <boost/graph/dijkstra_shortest_paths.hpp>
#include <boost/graph/filtered_graph.hpp>

typedef CGAL::Exact_predicates_inexact_constructions_kernel K;
typedef K::Point_2 Point;

typedef CGAL::Triangulation_vertex_base_with_id_2<K> Tvb;
typedef CGAL::Triangulation_face_base_2<K> Tfb;
typedef CGAL::Triangulation_data_structure_2<Tvb,Tfb> Tds;
typedef CGAL::Delaunay_triangulation_2<K, Tds> Triangulation;

// consider finite vertices and edges.

template <typename T>
struct Is_finite {

  const T* t_;

  Is_finite()
    : t_(NULL)
  {}

  Is_finite(const T& t)
    : t_(&t)
  { }

  template <typename VertexOrEdge>
  bool operator()(const VertexOrEdge& voe) const {
    return ! t_->is_infinite(voe);
  }
};

typedef Is_finite<Triangulation> Filter;
typedef boost::filtered_graph<Triangulation,Filter,Filter>
    Finite_triangulation;
typedef boost::graph_traits<Finite_triangulation>::vertex_descriptor
    vertex_descriptor;
typedef boost::graph_traits<Finite_triangulation>::vertex_iterator
    vertex_iterator;


int
```

```
main(int,char*[])
{
  Triangulation t;
  Filter is_finite(t);
  Finite_triangulation ft(t, is_finite, is_finite);

  t.insert(Point(0,0));
  t.insert(Point(1,0));
  t.insert(Point(0.2,0.2));
  t.insert(Point(0,1));
  t.insert(Point(0,2));

  vertex_iterator vit, ve;
  // associate indices to the vertices
  int index = 0;
  for(boost::tie(vit,ve)=boost::vertices(ft); vit!=ve; ++vit ){
    vertex_descriptor  vd = *vit;
    vd->id()= index++;
  }

  typedef boost::property_map<Triangulation, boost::vertex_index_t>::type
    VertexIdPropertyMap;
  VertexIdPropertyMap vertex_index_pmap = get(boost::vertex_index, ft);

  // Dijkstra's shortest path needs property maps for the predecessor and
    distance
  std::vector<vertex_descriptor> predecessor(boost::num_vertices(ft));
  boost::iterator_property_map<std::vector<vertex_descriptor>::iterator,
   VertexIdPropertyMap>
    predecessor_pmap(predecessor.begin(), vertex_index_pmap);

  std::vector<double> distance(boost::num_vertices(ft));
  boost::iterator_property_map<std::vector<double>::iterator,
   VertexIdPropertyMap>
    distance_pmap(distance.begin(), vertex_index_pmap);

  vertex_descriptor source =  *boost::vertices(ft).first;
  std::cout << "\nStart dijkstra_shortest_paths at " << source->point()
   << std::endl;

  boost::dijkstra_shortest_paths(ft, source ,
                                 distance_map(distance_pmap)
                                 .predecessor_map(predecessor_pmap));

  for(boost::tie(vit,ve)=boost::vertices(ft); vit!=ve; ++vit ){
    vertex_descriptor vd = *vit;
    std::cout << vd->point() << " [" << vd->id() << "] ";
    std::cout << " has distance = "  << get(distance_pmap,vd) << " and
   predecessor ";
    vd = get(predecessor_pmap,vd);
    std::cout << vd->point() << " [" << vd->id() << "]\n";
  }
  return 0;
}
```

File: examples/BGL_triangulation_2/dijkstra_with_internal_properties.cpp

74.6 Arrangements as Models of the Boost Graph Concept

For the arrangements CGAL offers the graph traits for the arrangement itself as well as for its dual graph.

74.6.1 Example for the Arrangement as Graph

Arrangement instances are adapted to *boost* graphs by specializing the *boost:graph_traits* template for *Arrangement_2* instances. The graph-traits states the graph concepts that the arrangement class models (see below) and defines the types required by these concepts.

In this specialization the *Arrangement_2* vertices correspond to the graph vertices, where two vertices are adjacent if there is at least one halfedge connecting them. More precisely, *Arrangement_2::Vertex_handle* is the graph-vertex type, while *Arrangement_2::Halfedge_handle* is the graph-edge type. As halfedges are directed, we consider the graph to be directed as well. Moreover, as several interior-disjoint x-monotone curves (say circular arcs) may share two common endpoints, inducing an arrangement with two vertices that are connected with several edges, we allow parallel edges in our *boost* graph.

Given an *Arrangement_2* instance, we can efficiently traverse its vertices and halfedges. Thus, the arrangement graph is a model of the concepts *VertexListGraph* and *EdgeListGraph* introduced by the BGL. At the same time, we use an iterator adapter of the circulator over the halfedges incident to a vertex (*Halfedge_around_vertex_circulator* — see Section 32.2.2), so it is possible to go over the ingoing and outgoing edges of a vertex in linear time. Thus, our arrangement graph is a model of the concept *BidirectionalGraph* (this concept refines *IncidenceGraph*, which requires only the traversal of outgoing edges).

It is important to notice that the vertex descriptors we use are *Vertex_handle* objects and *not* vertex indices. However, in order to gain more efficiency in most BGL algorithm, it is better to have them indexed $0,1,\ldots,(n-1)$, where n is the number of vertices. We therefore introduce the *Arr_vertex_index_map<Arrangement>* class-template, which maintains a mapping of vertex handles to indices, as required by the BGL. An instance of this class must be attached to a valid arrangement vertex when it is created. It uses the notification mechanism (see Section 32.7) to automatically maintain the mapping of vertices to indices, even when new vertices are inserted into the arrangement or existing vertices are removed.

In most algorithm provided by the BGL, the output is given by *property maps*, such that each map entry corresponds to a vertex. For example, when we compute the shortest paths from a given source vertex s to all other vertices we can obtain a map of distances and a map of predecessors — namely for each v vertex we have its distance from s and a descriptor of the vertex that precedes v in the shortest path from s. If the vertex descriptors are simply indices, one can use vectors to efficiently represent the property maps. As this is not the case with the arrangement graph, we offer the *Arr_vertex_property_map<Arrangement,Type>* template allows for an efficient mapping of *Vertex_handle* objects to properties of type *Type*. Note however that unlike the *Arr_vertex_index_map* class, the vertex property-map class is not kept synchronized with the number of vertices in the arrangement, so it should not be reused in calls to BGL functions in case the arrangement is modified in between these calls.

In the following example we construct an arrangement of 7 line segments, as shown in Figure 74.1, then use Dijkstra's shortest-paths algorithm from the BGL to compute the graph distance of all vertices from the leftmost vertex in the arrangement v_0. Note the usage of the *Arr_vertex_index_map* and the *Arr_vertex_property_map* classes. The latter one, instantiated by the type *double* is used to map vertices to their distances from v_0.

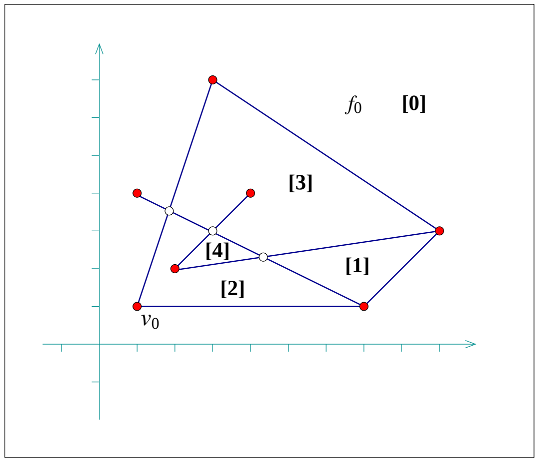


Figure 74.1: An arrangement of 7 line segments, as constructed by *ex_bgl_primal_adapter.cpp* and *ex_bgl_dual_adapter.cpp*. The breadth-first visit times for the arrangement faces, starting from the unbounded face f_0, are shown is brackets.

```
#include "arr_rational_nt.h"
#include <CGAL/Cartesian.h>
#include <CGAL/Arr_segment_traits_2.h>
#include <CGAL/Arrangement_2.h>

#include <climits>
#include <boost/graph/dijkstra_shortest_paths.hpp>

#include <CGAL/graph_traits_Arrangement_2.h>
#include <CGAL/Arr_vertex_index_map.h>

typedef CGAL::Cartesian<Number_type>                    Kernel;
typedef CGAL::Arr_segment_traits_2<Kernel>              Traits_2;
typedef Traits_2::Point_2                               Point_2;
typedef Traits_2::X_monotone_curve_2                    Segment_2;
typedef CGAL::Arrangement_2<Traits_2>                   Arrangement_2;

// A functor used to compute the length of an edge.
class Edge_length_func
{
public:

  // Boost property type definitions:
  typedef boost::readable_property_map_tag        category;
  typedef double                                  value_type;
  typedef value_type                              reference;
  typedef Arrangement_2::Halfedge_handle          key_type;

  double operator() (Arrangement_2::Halfedge_handle e) const
  {
    const double     x1 = CGAL::to_double (e->source()->point().x());
    const double     y1 = CGAL::to_double (e->source()->point().y());
    const double     x2 = CGAL::to_double (e->target()->point().x());
    const double     y2 = CGAL::to_double (e->target()->point().y());
```

```
    const double      diff_x = x2 - x1;
    const double      diff_y = y2 - y1;

    return (std::sqrt (diff_x*diff_x + diff_y*diff_y));
  }
};

double get (Edge_length_func edge_length, Arrangement_2::Halfedge_handle
    e)
{
  return (edge_length (e));
}

/* The folowing is a workaround for a bug in the BGL upto and including
    version
 * 103400.
 *
 * Unfortunately some of the calls to the get() function below from the
    BGL
 * code are qualified with the boost namespace, while others are not. For
     The
 * qualified calls the compiler naturally looks for the definition of the
 * function in boost namespace. For the other calls it searches the CGAL
 * namespace according to ADL (Koenig Lookup), as the type of the 1st
 * parameter is in CGAL namespace.
 *
 * One way to get around it is to provide 2 similar functions that do the
 * same thing. One in CGAL namespace provided in CGAL/Arr_vertex_map.h,
    and
 * the other in boost namespace below. The signature of the latter is
    slightly
 * changed to avoid redefinition. The type of its 1st parameter is
    defined in
 * boost namespace, and is a simple derivation of the 1st parameter of
    the
 * CGAL::get() function.
 */
namespace boost {

template <typename Arrangement_2>
class Arr_vertex_index_map_boost :
    public CGAL::Arr_vertex_index_map<Arrangement_2>
{
 public:
  typedef CGAL::Arr_vertex_index_map<Arrangement_2>     Base;
  /*! Default constructor. */
  Arr_vertex_index_map_boost() : Base() {}

  /*! Constructor from CGAL index map. */
  Arr_vertex_index_map_boost(Base & other) :
    CGAL::Arr_vertex_index_map<Arrangement_2>(other)
  {}
};
```

```
/*!
 * Get the index property-map function. Provided so that boost is able to
 * access the Arr_vertex_index_map above.
 * \param index_map The index map.
 * \param v A vertex handle.
 * \return The vertex index.
 */
template<class Arrangement>
unsigned int
get(const boost::Arr_vertex_index_map_boost<Arrangement> & index_map,
    typename Arrangement::Vertex_handle v)
{
  const CGAL::Arr_vertex_index_map<Arrangement> & index_map_tmp =
    static_cast<const CGAL::Arr_vertex_index_map<Arrangement>
   &>(index_map);
  return CGAL::get<Arrangement>(index_map_tmp, v);
}

}

int main ()
{
  Arrangement_2   arr;

  // Construct an arrangement of seven intersecting line segments.
  // We keep a handle for the vertex v_0 that corresponds to the point
   (1,1).
  Arrangement_2::Halfedge_handle  e =
    insert_non_intersecting_curve (arr, Segment_2 (Point_2 (1, 1),
                                                   Point_2 (7, 1)));
  Arrangement_2::Vertex_handle    v0 = e->source();
  insert (arr, Segment_2 (Point_2 (1, 1), Point_2 (3, 7)));
  insert (arr, Segment_2 (Point_2 (1, 4), Point_2 (7, 1)));
  insert (arr, Segment_2 (Point_2 (2, 2), Point_2 (9, 3)));
  insert (arr, Segment_2 (Point_2 (2, 2), Point_2 (4, 4)));
  insert (arr, Segment_2 (Point_2 (7, 1), Point_2 (9, 3)));
  insert (arr, Segment_2 (Point_2 (3, 7), Point_2 (9, 3)));

  // Create a mapping of the arrangement vertices to indices.
  CGAL::Arr_vertex_index_map<Arrangement_2>        index_map_tmp(arr);
  boost::Arr_vertex_index_map_boost<Arrangement_2>
   index_map(index_map_tmp);

  // Perform Dijkstra's algorithm from the vertex v0.
  Edge_length_func                              edge_length;
  CGAL::Arr_vertex_property_map<Arrangement_2,
                                double>        dist_map (index_map);

  boost::dijkstra_shortest_paths (arr, v0,
                                  boost::vertex_index_map (index_map).
                                  weight_map (edge_length).
                                  distance_map (dist_map));

  // Print the results:
```

```
  Arrangement_2::Vertex_iterator      vit;

  std::cout << "The distances of the arrangement vertices from ("
            << v0->point() << ") :" << std::endl;
  for (vit = arr.vertices_begin(); vit != arr.vertices_end(); ++vit)
  {
    std::cout << "(" << vit->point() << ") at distance "
              << dist_map[vit] << std::endl;
  }

  return (0);
}
```

File: examples/BGL_arrangement_2/primal.cpp

74.6.2 Example for the Dual of an Arrangement as Graph

It is possible to give a dual graph representation for an arrangement instance, such that each arrangement face corresponds to a graph vertex and two vertices are adjacent iff the corresponding faces share a common edge on their boundaries. This is done by specializing the *boost:graph_traits* template for *Dual<Arrangement_2>* instances, where *Dual<Arrangement_2>* is a template specialization that gives a dual interpretation to an arrangement instance.

In dual representation, *Arrangement_2::Face_handle* is the graph-vertex type, while *Arrangement_2::Halfedge_handle* is the graph-edge type. We treat the graph edges as directed, such that a halfedge e is directed from f_1, which is its incident face, to f_2, which is the incident face of its twin halfedge. As two arrangement faces may share more than a single edge on their boundary, we allow parallel edges in our *boost* graph. As is the case in the primal graph, the dual arrangement graph is also a model of the concepts *VertexListGraph*, *EdgeListGraph* and *BidirectionalGraph* (thus also of *IncidenceGraph*).

Since we use *Face_handle* objects as the vertex descriptors, we define the *Arr_face_index_map<Arrangement>* class-template, which maintains an efficient mapping of face handles to indices. We also provide the template *Arr_face_property_map<Arrangement,Type>* for associating arbitrary data with the arrangement faces.

In the following example we construct the same arrangement as in example *ex_bgl_primal_adapter.cpp* (see Figure 74.1), and perform breadth-first search on the graph faces, starting from the unbounded face. We extend the DCEL faces with an unsigned integer, marking the discover time of the face and use a breadth-first-search visitor to obtain these times and update the faces accordingly:

```
#include "arr_rational_nt.h"
#include <CGAL/Cartesian.h>
#include <CGAL/Arr_segment_traits_2.h>
#include <CGAL/Arr_extended_dcel.h>
#include <CGAL/Arrangement_2.h>

#include <climits>
#include <boost/graph/dijkstra_shortest_paths.hpp>

#include <CGAL/graph_traits_Dual_Arrangement_2.h>
#include <CGAL/Arr_face_index_map.h>
```

```
#include "arr_print.h"

typedef CGAL::Cartesian<Number_type>                        Kernel;
typedef CGAL::Arr_segment_traits_2<Kernel>                  Traits_2;
typedef Traits_2::Point_2                                   Point_2;
typedef Traits_2::X_monotone_curve_2                        Segment_2;
typedef CGAL::Arr_face_extended_dcel<Traits_2,
                                     unsigned int>          Dcel;
typedef CGAL::Arrangement_2<Traits_2, Dcel>                 Arrangement_2;
typedef CGAL::Dual<Arrangement_2>
   Dual_arrangement_2;

// A BFS visitor class that associates each vertex with its discover
   time.
// In our case graph vertices represent arrangement faces.
template <class IndexMap>
class Discover_time_bfs_visitor : public boost::default_bfs_visitor
{
private:

  const IndexMap  *index_map;      // Mapping vertices to indices.
  unsigned int     time;           // The current time stamp.

public:

  // Constructor.
  Discover_time_bfs_visitor (const IndexMap& imap) :
    index_map (&imap),
    time (0)
  {}

  // Write the discover time for a given vertex.
  template <typename Vertex, typename Graph>
  void discover_vertex (Vertex u, const Graph& )
  {
    u->set_data (time);
    time++;
  }
};

int main ()
{
  Arrangement_2   arr;

  // Construct an arrangement of seven intersecting line segments.
  insert (arr, Segment_2 (Point_2 (1, 1), Point_2 (7, 1)));
  insert (arr, Segment_2 (Point_2 (1, 1), Point_2 (3, 7)));
  insert (arr, Segment_2 (Point_2 (1, 4), Point_2 (7, 1)));
  insert (arr, Segment_2 (Point_2 (2, 2), Point_2 (9, 3)));
  insert (arr, Segment_2 (Point_2 (2, 2), Point_2 (4, 4)));
  insert (arr, Segment_2 (Point_2 (7, 1), Point_2 (9, 3)));
  insert (arr, Segment_2 (Point_2 (3, 7), Point_2 (9, 3)));

  // Create a mapping of the arrangement faces to indices.
```

```
  CGAL::Arr_face_index_map<Arrangement_2>      index_map (arr);

  // Perform breadth-first search from the unbounded face, and use the
    BFS
  // visitor to associate each arrangement face with its discover time.
  Discover_time_bfs_visitor<CGAL::Arr_face_index_map<Arrangement_2> >
                                               bfs_visitor (index_map);
  Arrangement_2::Face_handle                   uf = arr.unbounded_face();

  boost::breadth_first_search (Dual_arrangement_2 (arr), uf,
                               boost::vertex_index_map (index_map).
                               visitor (bfs_visitor));

  // Print the results:
  Arrangement_2::Face_iterator      fit;

  for (fit = arr.faces_begin(); fit != arr.faces_end(); ++fit)
  {
    std::cout << "Discover time " << fit->data() << " for ";
    if (fit != uf)
    {
      std::cout << "face ";
      print_ccb<Arrangement_2> (fit->outer_ccb());
    }
    else
      std::cout << "the unbounded face." << std::endl;
  }

  return (0);
}
```

File: examples/BGL_arrangement_2/dual.cpp

CGAL and the Boost Graph Library Reference Manual

Andreas Fabri, Fernando Cacciola, and Ron Wein

This chapter introduces the concepts and classes that allow to interface CGAL data structures with the algorithms of the BGL, as well as a new graph concept.

74.7 Classified Reference Pages

Concepts

Enums

Classes

74.8 Alphabetical List of Reference Pages

HalfedgeGraph

Concept

Definition

The concept HalfedgeGraph describes the requirements for a graph that is structurally equivalent to a polyhedral surface represented by a halfedge data structure, and it provides an interface for efficient access to the opposite edge of an edge, and to the successor and predecessor of an edge in the iterator range of the incoming edges of a vertex. Each vertex has a geometric position in space. As in a halfedge data structure we define the face adjacent to a halfedge to be to the *left* of the halfedge.

Requirements

For each *directed edge* $e = (v,w)$ its opposite edge $e' = (w,v)$ must be part of the graph.

The incoming edges of a vertex v have a fixed order, that is all calls of *in_edges(v,g)* must return the same iterator range, modulo a cyclic permutation. The order must be *clockwise*.

As the HalfedgeGraph is equivalent to a polyhedral surface there must exist an embedding for the vertices and edges such that the ordered edges do not intersect.

Refines

BidirectionalGraph
PropertyGraph

A model of HalfedgeGraph must have the *interior properties edge_is_border* attached to its edges, and it must have *vertex_is_border* and *vertex_point* attached to its vertices.

Associated Types

HalfedgeGraph:: halfedge_graph_traits<HalfedgeGraph>::Point

The type of the geometric location of a vertex.

Because (directed) edges must come in pairs, there is the additional notion of an *undirected edge*[1] for a pair of opposite directed edges. The number of undirected edges is exactly half the number of directed edges.

Note that the notion of directed and undirected edges does not imply the existence of two different types. The type *edge_descriptor* is used for both. An undirected edge must be implicitly handled, and there is no requirement on which of the directed edges of the undirected edge must be used to represent it.

HalfedgeGraph:: halfedge_graph_traits<HalfedgeGraph>::undirected_edge_iterator

An iterator that iterates over one and only one of the directed edges in each pair of opposite directed edges. The value type of the iterator is *boost::graph_traits<HalfedgeGraph> ::edge_descriptor*.

[1] The directed edges are not called *halfedges* (as in a *HalfedgeDS*) because from the point of view of this graph, being a refinement of a BGL graph, each directed edge is an edge in itself. In other words, the unqualified term edge refers to one and only one directed edge and not to a pair.

Valid Expressions

Following the BGL design, the following graph operations are defined as free rather than member functions.

template<class Graph>
std::pair<typename halfedge_graph_traits<HalfedgeGraph>::undirected_edge_iterator, typename halfedge_graph_traits<HalfedgeGraph>::undirected_edge_iterator>

undirected_edges(Graph g)

Returns the undirected edges of g.

An edge $e = (v, w)$ is said to be the *opposite edge* of edge $e' = (w, v)$.

template<class Graph>
typename boost::graph_traits<Graph const>::edge_descriptor

opposite_edge(typename boost::graph_traits<Graph const>::edge_descriptor e, Graph g)

Returns the opposite edge of e.

An edge $e' = (v, w)$ is called the *clockwise neighbor* of edge $e = (u, w)$, and e the *counterclockwise neighbor* of e', iff there exist two iterators *it* and it' in the iterator range *in_edges(w,g)* such that ***it* == *e* and ***it' == e'*, and *it' == it*++ or *it* is the last and *it'* the first iterator of the iterator range.

template<class Graph>
typename boost::graph_traits<Graph const>::edge_descriptor

next_edge_cw(typename boost::graph_traits<Graph const>::edge_descriptor e, Graph g)

Returns the clockwise neighbor of e.

template<class Graph>
typename boost::graph_traits<Graph const>::edge_descriptor

next_edge_ccw(typename boost::graph_traits<Graph const>::edge_descriptor e, Graph g)

Returns the counterclockwise neighbor of e.

A composition of these access functions yields an access function for the edge cycle adjacent to the same face. An edge $e' = (v, w)$ is called the *successor* of edge $e = (u, v)$, and e the *predecessor* of e', iff e' is the clockwise neighbor of the opposite edge of e.

template<class Graph>
typename boost::graph_traits<Graph const>::edge_descriptor

next_edge(typename boost::graph_traits<Graph const>::edge_descriptor e, Graph g)

Returns the successor of e.

template<class Graph>
typename boost::graph_traits<Graph const>::edge_descriptor

prev_edge(typename boost::graph_traits<Graph const>::edge_descriptor e, Graph g)

Returns the predecessor of *e*.

Has Models

CGAL::Polyhedron_3<Traits> . . . . page 1795

Class

boost::graph_traits< CGAL::Arrangement_2<Traits,Dcel> >

Definition

The class *graph_traits< CGAL::Arrangement_2<Traits,Dcel> >* is a partial specialization of *boost::graph_traits* for the class *Arrangement_2*. It provides the types associated to the graph concepts *BidirectionalGraph* and *EdgeAndVertexListGraph*.

#include <CGAL/boost/graph/graph_traits_Arrangement_2.h>

Types

typename CGAL::Arrangement_2::Vertex_handle

vertex_descriptor; The vertex descriptor.

typename CGAL::Arrangement_2::Halfedge_handle

edge_descriptor; The edge descriptor.

graph_traits< CGAL::Arrangement_2<Traits,Dcel> >:: vertex_iterator

An iterator corresponding to *CGAL::Arrangement_2::Vertex_iterator*, with the difference that its value type is a vertex descriptor and not *CGAL::Arrangement_2::Vertex*.

graph_traits< CGAL::Arrangement_2<Traits,Dcel> >:: edge_iterator

An iterator corresponding to *CGAL::Arrangement_2::Halfedge_iterator* with the difference that its value type is an edge descriptor and not *CGAL::Arrangement_2::Halfedge*.

graph_traits< CGAL::Arrangement_2<Traits,Dcel> >:: in_edge_iterator

An edge iterator which only iterates over the incoming edges around a vertex. It corresponds to a *CGAL::Arrangement_2::Halfedge_around_vertex_circulator* with the difference that its value type is an edge descriptor and not *CGAL::Arrangement_2::Halfedge*.

graph_traits< CGAL::Arrangement_2<Traits,Dcel> >:: out_edge_iterator

An edge iterator which only iterates over the outgoing halfedges around a vertex. It corresponds to a *CGAL::Arrangement_2::Halfedge_around_vertex_circulator* with the difference that its value type is an edge descriptor and not *CGAL::Arrangement_2::Halfedge*.

boost::sallow_parallel_edge_tag

edge_parallel_category;

Indicates that this graph does support multiedges.

boost::bidirectional_graph_tag

traversal_category; Indicates that this graph is bidirectional.

typename Arrangement_2::size_type

vertices_size_type; The size type of the vertex list.

typename Arrangement_2::size_type

edges_size_type; The size type of the edge list.

typename Arrangement_2::size_type

degree_size_type; The size type of the adjacency list.

The const specialization, *boost::graph_traits*< *CGAL::Arrangement_2*<*Traits,Dcel*> *const*> is also defined, using the constant handles in the arrangement.

Class

boost::graph_traits< CGAL::Polyhedron_3<Traits> >

Definition

The class *graph_traits< CGAL::Polyhedron_3<Traits> >* is a partial specialization of *boost::graph_traits* for the class *Polyhedron_3*. It provides the types associated to the graph concepts *BidirectionalGraph* and *EdgeAndVertexListGraph*.

#include <CGAL/boost/graph/graph_traits_Polyhedron_3.h>

Types

typename CGAL::Polyhedron_3::Vertex_handle

vertex_descriptor; The vertex descriptor.

typename CGAL::Polyhedron_3::Halfedge_handle

edge_descriptor; The edge descriptor.

graph_traits< CGAL::Polyhedron_3<Traits> >:: vertex_iterator

An iterator corresponding to *CGAL::Polyhedron_3::Vertex_iterator*, with the difference that its value type is a vertex descriptor and not *CGAL::Polyhedron_3::Vertex*.

graph_traits< CGAL::Polyhedron_3<Traits> >:: edge_iterator

An iterator corresponding to *CGAL::Polyhedron_3::Halfedge_iterator* with the difference that its value type is an edge descriptor and not *CGAL::Polyhedron_3::Halfedge*.

graph_traits< CGAL::Polyhedron_3<Traits> >:: in_edge_iterator

An edge iterator which only iterates over the incoming edges around a vertex. It corresponds to a *CGAL::Polyhedron_3::Halfedge_around_vertex_circulator* with the difference that its value type is an edge descriptor and not *CGAL::Polyhedron_3::Halfedge*.

graph_traits< CGAL::Polyhedron_3<Traits> >:: out_edge_iterator

An edge iterator which only iterates over the outgoing halfedges around a vertex. It corresponds to a *CGAL::Polyhedron_3::Halfedge_around_vertex_circulator* with the difference that its value type is an edge descriptor and not *CGAL::Polyhedron_3::Halfedge*.

boost::disallow_parallel_edge_tag

edge_parallel_category;

Indicates that this graph does not support multiedges.

boost::bidirectional_graph_tag

traversal_category; Indicates that this graph is bidirectional.

typename Polyhedron_3::size_type

vertices_size_type; The size type of the vertex list.

typename Polyhedron_3::size_type

edges_size_type; The size type of the edge list.

typename Polyhedron_3::size_type

degree_size_type; The size type of the adjacency list.

The const specialization, *boost::graph_traits*< *CGAL::Polyhedron_3*<*Traits*> *const*> is also defined, using the constant handles in the polyhedron.

Class

boost::graph_traits< CGAL::Triangulation_2<GT, TDS> >

Definition

The triangulations of CGAL are all models of the concepts *BidirectionalGraph* and *VertexAndEdgeListGraph* of the Boost Graph Library [SLL02].

The class *graph_traits< CGAL::Triangulation_2<GT, TDS> >* is a partial specialization of the class *boost::graph_traits<G>*.

The mapping between vertices and edges of the triangulation and the graph is rather straightforward, but there are some subtleties. The value type of the BGL iterators is the vertex or edge descriptor, whereas in CGAL all iterators and circulators are also handles and hence have as value type Vertex or Edge.

The graph traits class for triangulations does not distinguish between finite and infinite vertices and edges. As the edge weight computed with the default property map of BGL algorithms (obtained with *boost::get(t, boost::edge_weight)*) is the length of the edge, the edge weight is not well defined for infinite edges. For algorithms that make use of the edge weight the user must therefore define a *boost::filtered_graph* or pass a property map to the algorithm that returns "infinity" for infinite edges.

Note also that when you derive from the class *CGAL::Triangulation_2* you must upcast the object in order to use this partial specialization.

For the user convenience, CGAL provides the partial specializations for all 2D triangulation classes.

#include <CGAL/boost/graph/graph_traits_Triangulation_2.h>

Types

typedef Triangulation::Vertex_handle

vertex_descriptor; The vertex descriptor.

graph_traits< CGAL::Triangulation_2<GT, TDS> >:: edge_descriptor

The edge descriptor. It is constructible from and convertible to *Triangulation::Edge*. The edge descriptor is not a simple typedef, but a proper class, because in an undirected graph the edges *(u,v)* and *(v,u)* must be equal. This is not the case for the Edge type of the triangulation.

graph_traits< CGAL::Triangulation_2<GT, TDS> >:: vertex_iterator

The vertex iterator type. Its value type is *vertex_descriptor*.

graph_traits< CGAL::Triangulation_2<GT, TDS> >:: edge_iterator

The edge iterator type, Its value type is *edge_descriptor*.

graph_traits< CGAL::Triangulation_2<GT, TDS> >:: out_edge_iterator

An iterator for the outgoing edges incident to a vertex. Its value type is *edge_descriptor*.

graph_traits< CGAL::Triangulation_2<GT, TDS> >:: in_edge_iterator

An iterator for the incoming edges incident to a vertex. Its value type is *edge_descriptor*.

graph_traits< CGAL::Triangulation_2<GT, TDS> >:: adjacency_iterator

An iterator for the vertices adjacent to a vertex. Its value type is *vertex_descriptor*.

typedef boost::undirected_tag

directed_category;

typedef boost::disallow_parallel_edge_tag

edge_parallel_category;

Class

CGAL::halfedge_graph_traits< CGAL::Polyhedron_3<Traits> >

Definition

The class *halfedge_graph_traits< CGAL::Polyhedron_3<Traits> >* is a partial specialization of *CGAL::halfedge_graph_traits* for *Polyhedron_3*. It provides the types associated to the *HalfedgeGraph* concept.

#include <CGAL/boost/graph/halfedge_graph_traits_Polyhedron_3.h>

Types

halfedge_graph_traits< CGAL::Polyhedron_3<Traits> >:: undirected_edge_iterator

An edge iterator that iterates over one of the two opposite edges forming an undirected edge.
The value type is *CGAL::Polyhedron_3::Halfedge_const_handle*.

typename CGAL::Polyhedron_3<Traits>::Point_3

Point;	The point type of the vertex.

CGAL::Triangulation_vertex_base_with_id_2<TriangulationTraits_2, TriangulationVertexBase_2>

Class

Definition

The class *Triangulation_vertex_base_with_id_2* is a model of the concept *TriangulationVertexBase_2*, the base vertex of a 2D-triangulation. It provides an integer field that can be used to index vertices for BGL algorithms.

Note that the user is in charge to set the index correctly before running a graph algorithm.

#include <CGAL/Triangulation_vertex_base_with_id_2.h>

Parameters

The first template argument is the geometric traits class *TriangulationTraits_2* which provides the *Point_2*.

The second template argument is a vertex base class from which *Triangulation_vertex_base_with_id_2* derives. It has the default value *Triangulation_vertex_base_2<TriangulationTraits_2>*.

Is Model for the Concepts

TriangulationVertexBase_2

Inherits From

TriangulationVertexBase_2

Access Functions

int	*v.id() const*	Returns the index.
int&	*v.id()*	Returns a reference to the index stored in the vertex.

See Also

CGAL::Triangulation_vertex_base_2

CGAL::HalfedgeDS_halfedge_max_base_with_id<Refs>

Definition

The class *HalfedgeDS_halfedge_max_base_with_id<Refs>* is a model of the *HalfedgeDSHalfedge* concept. *Refs* is an instantiation of a *HalfedgeDS*. It is equivalent to *CGAL::HalfedgeDS_halfedge_base< Refs, CGAL::Tag_true, CGAL::Tag_true, CGAL::Tag_true>* with an added integer field which can be used to index halfedges in BGL algorithms. The class contains support for the previous, next, opposite, vertex and face pointers and the required type definitions. It can be used for deriving own halfedges.

Note that the user is in charge to set the index correctly before running a graph algorithm.

#include <CGAL/HalfedgeDS_halfedge_max_base_with_id.h>

Is Model for the Concepts

HalfedgeDSHalfedge . . . . page 1852

Creation

HalfedgeDS_halfedge_max_base_with_id<Refs> h;

default constructor.

int	*h.id() const*	Returns the index.
int&	*h.id()*	Returns a reference to the index stored in the halfedge.

See Also

HalfedgeDS<Traits,Items,Alloc> . . . . page 1843
HalfedgeDSItems . . . . page 1854
PolyhedronItems_3 . . . . page 1818
CGAL::HalfedgeDS_min_items . . . . page 1879
CGAL::HalfedgeDS_vertex_min_base<Refs> . . . . page 1885
CGAL::HalfedgeDS_face_min_base<Refs> . . . . page 1870
CGAL::HalfedgeDS_halfedge_base<Refs> . . . . page 1871
CGAL::HalfedgeDS_vertex_max_base_with_id<Refs> . . . . page 4290
CGAL::HalfedgeDS_face_max_base_with_id<Refs> . . . . page 4289
CGAL::Polyhedron_items_with_id_3 . . . . page 4291

CGAL::HalfedgeDS_face_max_base_with_id<Refs>

Definition

The class *HalfedgeDS_face_max_base_with_id<Refs>* is a model of the *HalfedgeDSFace* concept. *Refs* is an instantiation of a *HalfedgeDS*. It is equivalent to *CGAL::HalfedgeDS_face_base< Refs, CGAL::Tag_true>* with an added integer field which can be used to index faces in BGL algorithms.. The class contains support for the incident halfedge pointer and the required type definitions. It can be used for deriving own faces.

Note that the user is in charge to set the index correctly before running a graph algorithm.

#include <CGAL/HalfedgeDS_face_max_base_with_id.h>

Is Model for the Concepts

HalfedgeDSFace . . . . page 1850

Creation

HalfedgeDS_face_max_base_with_id<Refs> f;		default constructor.
int	*f.id() const*	Returns the index.
int&	*f.id()*	Returns a reference to the index stored in the face.

See Also

HalfedgeDS<Traits,Items,Alloc> . . . . page 1843
HalfedgeDSItems . . . . page 1854
PolyhedronItems_3 . . . . page 1818
CGAL::HalfedgeDS_min_items . . . . page 1879
CGAL::HalfedgeDS_vertex_min_base<Refs> . . . . page 1885
CGAL::HalfedgeDS_halfedge_min_base<Refs> . . . . page 1872
CGAL::HalfedgeDS_face_base<Refs> . . . . page 1868
CGAL::HalfedgeDS_vertex_max_base_with_id<Refs> . . . . page 4290
CGAL::HalfedgeDS_halfedge_max_base_with_id<Refs> . . . . page 4288
CGAL::Polyhedron_items_with_id_3 . . . . page 4291

CGAL::HalfedgeDS_vertex_max_base_with_id<Refs>

Definition

The class *HalfedgeDS_vertex_max_base_with_id<Refs>* is a model of the *HalfedgeDSVertex* concept. *Refs* is an instantiation of a *HalfedgeDS*. It is equivalent to *CGAL::HalfedgeDS_vertex_base< Refs, CGAL::Tag_true>* with an added integer field which can be used to index vertices in BGL algorithms.. The class contains support for the point and the required type definitions. It can be used for deriving own vertices.

Note that the user is in charge to set the index correctly before running a graph algorithm.

#include <CGAL/HalfedgeDS_vertex_max_base_with_id.h>

Is Model for the Concepts

HalfedgeDSVertex . . . . page 1856

Creation

HalfedgeDS_vertex_max_base_with_id<Refs> v;		default constructor.
int	*v.id() const*	Returns the index.
int&	*v.id()*	Returns a reference to the index stored in the vertex.

See Also

HalfedgeDS<Traits,Items,Alloc> . . . . page 1843
HalfedgeDSItems . . . . page 1854
PolyhedronItems_3 . . . . page 1818
CGAL::HalfedgeDS_min_items . . . . page 1879
CGAL::HalfedgeDS_halfedge_min_base<Refs> . . . . page 1872
CGAL::HalfedgeDS_face_min_base<Refs> . . . . page 1870
CGAL::HalfedgeDS_vertex_base<Refs> . . . . page 1883
CGAL::HalfedgeDS_halfedge_max_base_with_id<Refs> . . . . page 4288
CGAL::HalfedgeDS_face_max_base_with_id<Refs> . . . . page 4289
CGAL::Polyhedron_items_with_id_3 . . . . page 4291

CGAL::Polyhedron_items_with_id_3

Definition

The class *Polyhedron_items_with_id_3* is a model of the *PolyhedronItems_3* concept. It provides definitions for vertices with points, halfedges, and faces with plane equations, all of them with an additional integer field which can be used to index the items in a BGL algorithm. The polyhedron traits class must provide the respective types for the point and the plane equation. Vertices and facets both contain a halfedge handle to an incident halfedge.

#include <CGAL/Polyhedron_items_with_id_3.h>

Is Model for the Concepts

PolyhedronItems_3 . . . . page 1818

Types in *Polyhedron_items_with_id_3::Vertex_wrapper<Refs,Traits>::Vertex*

typedef Traits::Point_3 *Point;*
typedef CGAL::Tag_true *Supports_vertex_point;*

Types in *Polyhedron_items_with_id_3::Face_wrapper<Refs,Traits>::Face*

typedef Traits::Plane_3 *Plane;*
typedef CGAL::Tag_true *Supports_face_plane;*

Additional methods in all three items.

int *item.id() const*

Returns the index.

int& *item.id()* Returns a reference to the index stored in the item.

Creation

Polyhedron_items_with_id_3 items; default constructor.

Operations

Supported as required by the *PolyhedronItems_3* concept.

See Also

CGAL::Polyhedron_3<Traits> . . . . page 1795
CGAL::Polyhedron_min_items_3 . . . . page 1822

boost::edge_index_t

enum edge_index_t { *edge_index*}*;*

The constant *edge_index* is a property tag which identifies the *index* property of an edge of a BGL Graph.

Enum

CGAL::edge_is_border_t

#include <CGAL/boost/graph/properties.h>

enum edge_is_border_t { *edge_is_border* }*;*

The constant *edge_is_border* is a property tag which refers to the property of an edge of being a border edge.

edge_is_border is an interior property. That is, a property map for *edge_is_border* can be extracted from any model of a *HalfedgeGraph* using the BGL PropertyGraph interface:

boost::get(edge_is_border,graph)

The Boolean flag that indicates if the edge is a border can be directly accessed via:

boost::get(edge_is_border,graph,edge).

CGAL::vertex_is_border_t

#include <CGAL/boost/graph/properties.h>

enum vertex_is_border_t { *vertex_is_border*}*;*

The constant *vertex_is_border* is a property tag which refers to the property of a vertex of being a border vertex.

vertex_is_border is an interior property, that is, a property map for *vertex_is_border* can be extracted from any model of a *HalfedgeGraph* using the BGL PropertyGraph interface:

boost::get(vertex_is_border,graph)

The Boolean flag that indicates if the vertex is a border can be directly accessed via:

boost::get(vertex_is_border,graph,edge)

Enum

CGAL::vertex_point_t

#include <CGAL/boost/graph/properties.h>

enum vertex_point_t { *vertex_point*}*;*

The constant *vertex_point* is a property tag which refers to the geometric embedding property of a vertex of a *HalfedgeGraph*.

A *vertex_point* is an interior property, that is, a property map for a *vertex_point* can be extracted from any model of a *HalfedgeGraph* using the BGL PropertyGraph interface:

boost::get(vertex_point,graph)

A point of a vertex can be directly accessed via:

boost::get(vertex_point,graph,vertex)
boost::put(vertex_point,graph,vertex,newpoint)

Chapter 75

CGAL and Boost Property Maps

Andreas Fabri and Laurent Saboret

Contents

75.1 A Short Introduction to the Boost Property Maps Library

The Boost Property Map Library consists mainly of interface specifications in the form of concepts. These interface specifications are intended for use by implementors of generic libraries in communicating requirements on template parameters to their users. In particular, the Boost Property Map concepts define a general purpose interface for mapping key objects to corresponding value objects, thereby hiding the details of how the mapping is implemented from algorithms. The implementation of types fulfilling the property map interface is up to the client of the algorithm to provide.

The Boost Property Map Library also contains a few adaptors that convert commonly used data-structures that implement a mapping operation, such as builtin arrays (pointers), iterators, and std::map, to have the property map interface.

Free functions *get* and *put* allow getting and putting information through a property map. The data themselves may be stored in the element, or they may be stored in an external data structure, or they may be computed on the fly. This is an “implementation detail” of the particular property map.

Property maps in the Boost manuals: `http://www.boost.org/libs/property_map/doc/property_map.html`

75.2 CGAL and Boost Property Maps

Some algorithms in CGAL take as input parameters iterator ranges and property maps to access information attached to elements of the sequence.

For example, the algorithms of chapters *Point_set_processing_3* 58 and *Surface_reconstruction_points_3* 49 take as input parameters iterator ranges and property maps to access each point's position and normal. Position and normal might be represented in various ways, e.g., as a class derived from the CGAL point class, or as a *std::pair<Point_3<K>, Vector_3<K> >*, or as a *boost::tuple<..,Point_3<K>, ..., Vector_3<K> >*.

This component provides property maps to support these cases:
CGAL::Dereference_property_map<T>
CGAL::First_of_pair_property_map<Pair> and *CGAL::Second_of_pair_property_map<Pair>*
CGAL::Nth_of_tuple_property_map<N, Tuple>

75.2.1 Example with Dereference_property_map

The following example reads a point set and removes 5% of the points. It uses *CGAL::Dereference_property_map<Point_3>* as position property map.

```
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/property_map.h>
#include <CGAL/remove_outliers.h>
#include <CGAL/IO/read_xyz_points.h>

#include <vector>
#include <fstream>

// types
typedef CGAL::Exact_predicates_inexact_constructions_kernel Kernel;
typedef Kernel::Point_3 Point;

int main(void)
{
  // Reads a .xyz point set file in points[].
  // The Dereference_property_map property map can be omitted here as it
   is the default value.
  std::vector<Point> points;
  std::ifstream stream("data/oni.xyz");
  if (!stream ||
      !CGAL::read_xyz_points(stream, std::back_inserter(points),
                             CGAL::Dereference_property_map<Point>()))
  {
    std::cerr << "Error: cannot read file data/oni.xyz" << std::endl;
    return EXIT_FAILURE;
  }

  // Removes outliers using erase-remove idiom.
  // The Dereference_property_map property map can be omitted here as it
   is the default value.
  const double removed_percentage = 5.0; // percentage of points to
   remove
```

```
  const int nb_neighbors = 24; // considers 24 nearest neighbor points
  points.erase(CGAL::remove_outliers(points.begin(), points.end(),

   CGAL::Dereference_property_map<Point>(),
                                               nb_neighbors, removed_percentage),
                points.end());

  // Optional: after erase(), use Scott Meyer's "swap trick" to trim
    excess capacity
  std::vector<Point>(points).swap(points);

  return EXIT_SUCCESS;
}
```

File: examples/Point_set_processing_3/remove_outliers_example.cpp

75.2.2 Example with Pairs

The following example reads a point set from an input file and writes it to a file, both in the xyz format. Position and normal are stored in pairs and accessed through property maps.

```
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/property_map.h>
#include <CGAL/IO/read_xyz_points.h>
#include <CGAL/IO/write_xyz_points.h>

#include <utility> // defines std::pair
#include <vector>
#include <fstream>

// types
typedef CGAL::Exact_predicates_inexact_constructions_kernel Kernel;
typedef Kernel::Point_3 Point;
typedef Kernel::Vector_3 Vector;

// Point with normal vector stored as a std::pair.
typedef std::pair<Point, Vector> Pwn;

int main(void)
{
    // Reads a .xyz point set file in points[].
    // Note: read_xyz_points_and_normals() requires an output iterator
    // over points and as well as property maps to access each
    // point position and normal.
    std::vector<Pwn> points;
    std::ifstream in("data/oni.xyz");
    if (!in ||
        !CGAL::read_xyz_points_and_normals(
            in,std::back_inserter(points),
            CGAL::First_of_pair_property_map<Pwn>(),
            CGAL::Second_of_pair_property_map<Pwn>()))
    {
```

```
      std::cerr << "Error: cannot read file data/oni.xyz" << std::endl;
      return EXIT_FAILURE;
    }

    // Saves point set.
    // Note: write_xyz_points_and_normals() requires an output iterator
    // over points as well as property maps to access each
    // point position and normal.
    std::ofstream out("oni_copy.xyz");
    if (!out ||
        !CGAL::write_xyz_points_and_normals(
            out, points.begin(), points.end(),
            CGAL::First_of_pair_property_map<Pwn>(),
            CGAL::Second_of_pair_property_map<Pwn>()))
    {
      return EXIT_FAILURE;
    }

    return EXIT_SUCCESS;
}
```

File: examples/Point_set_processing_3/read_write_xyz_point_set_example.cpp

75.2.3 Example with Tuples

The following example reads a point set in the *xyz* format and computes the average spacing. Index, position and color are stored in a tuple and accessed through property maps.

```
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/compute_average_spacing.h>
#include <CGAL/IO/read_xyz_points.h>

#include <vector>
#include <fstream>
#include <boost/tuple/tuple.hpp>

// Types
typedef CGAL::Exact_predicates_inexact_constructions_kernel Kernel;
typedef Kernel::FT FT;
typedef Kernel::Point_3 Point;

// Data type := index, followed by the point, followed by three integers
    that
// define the Red Green Blue color of the point.
typedef boost::tuple<int, Point, int, int, int>
    IndexedPointWithColorTuple;

int main(void)
{
    // Reads a .xyz point set file in points.
    // As the point is the second element of the tuple (that is with
    index 1)
```

```
    // we use a property map that accesses the 1st element of the tuple.
    std::vector<IndexedPointWithColorTuple> points;
    std::ifstream stream("data/sphere_20k.xyz");
    if (!stream ||
        !CGAL::read_xyz_points(
            stream, std::back_inserter(points),

   CGAL::Nth_of_tuple_property_map<1,IndexedPointWithColorTuple>()))
    {
      std::cerr << "Error: cannot read file data/sphere_20k.xyz" <<
   std::endl;
      return EXIT_FAILURE;
    }

    // Initialize index and RGB color fields in tuple.
    // As the index and RGB color are respectively the first and
   third-fifth elements
    // of the tuple we use a get function from the property map that
   accesses the 0
    // and 2-4th elements of the tuple.
    for(unsigned int i = 0; i < points.size(); i++)
    {
      points[i].get<0>() = i;   // set index value of tuple to i

      points[i].get<2>() = 0;   // set RGB color to black
      points[i].get<3>() = 0;
      points[i].get<4>() = 0;
    }

    // Computes average spacing.
    const unsigned int nb_neighbors = 6; // 1 ring
    FT average_spacing = CGAL::compute_average_spacing(
                          points.begin(), points.end(),

   CGAL::Nth_of_tuple_property_map<1,IndexedPointWithColorTuple>(),
                          nb_neighbors);
    std::cout << "Average spacing: " << average_spacing << std::endl;

    return EXIT_SUCCESS;
}
```

File: examples/Point_set_processing_3/average_spacing_example.cpp

CGAL and Boost Property Maps Reference Manual

Andreas Fabri and Laurent Saboret

This package provides a framework for interfacing CGAL data structures with algorithms expecting Boost Property Maps.

75.3 Classified Reference Pages

Classes

75.4 Alphabetical List of Reference Pages

Class

CGAL::Dereference_property_map<T>

Definition

Property map that converts a *T** pointer (or in general an iterator over *T* elements) to the *T* object.

#include <CGAL/property_map.h>

Parameters

template<typename T>
struct *Dereference_property_map*;

Inherits From

boost::put_get_helper< T&, Dereference_property_map<T> >

Is Model for the Concepts

Model of *boost::LvaluePropertyMap* concept.

Types

Dereference_property_map<T>:: key_type	typedef to *T**
Dereference_property_map<T>:: value_type	typedef to *T*
Dereference_property_map<T>:: reference	typedef to *T&*
Dereference_property_map<T>:: category	*boost::lvalue_property_map_tag*

Creation

Dereference_property_map<T> pmap; — Constructor.

Operations

template<class Iter>
reference *pmap*[*Iter it*] — Access a property map element.
Template Parameters:
Iter: Type convertible to *key_type*.

Related Functions

Dereference_property_map<typename value_type_traits<Iter>::type>

make_dereference_property_map(Iter)

Free function to create a *Dereference_property_map* property map.

Example

See *property_map.cpp* example.

CGAL::First_of_pair_property_map<Pair>

Definition

Property map that accesses the first item of a *std::pair*.

#include <CGAL/property_map.h>

Parameters

template<typename Pair>
struct *First_of_pair_property_map*;

Parameters:
Pair: Instance of *std::pair*.

Inherits From

boost::put_get_helper< Pair::first_type& , First_of_pair_property_map<Pair> >

Is Model for the Concepts

Model of *boost::LvaluePropertyMap* concept.

Types

First_of_pair_property_map<Pair>:: key_type	typedef to *Pair**
First_of_pair_property_map<Pair>:: value_type	typedef to *Pair::first_type*
First_of_pair_property_map<Pair>:: reference	typedef to *value_type&*
First_of_pair_property_map<Pair>:: category	*boost::lvalue_property_map_tag*

Creation

First_of_pair_property_map<Pair> pmap;	Constructor.

Operations

template<class Iter> *reference*	*pmap*[*Iter pair*]	Access a property map element. *Template Parameters*: *Iter*: Type convertible to *key_type*.

Related Functions

First_of_pair_property_map<typename value_type_traits<Iter>::type>

make_first_of_pair_property_map(Iter)

Free function to create a *First_of_pair_property_map* property map.

See Also

CGAL::Second_of_pair_property_map<Pair> . . . . page 4309

Example

See *property_map.cpp* example.

CGAL::Nth_of_tuple_property_map<N, Tuple>

Class

Definition

Property map that accesses the Nth item of a *boost::tuple*.

#include <CGAL/property_map.h>

Parameters

template<int N, typename Tuple>
struct *Nth_of_tuple_property_map*;

Parameters:
N: Index of the item to access. *Tuple*: Instance of *boost::tuple*.

Inherits From

boost::put_get_helper< boost::tuples::element<N, Tuple>::type& , *Nth_of_tuple_property_map<N, Tuple> >*

Is Model for the Concepts

Model of *boost::LvaluePropertyMap* concept.

Types

Nth_of_tuple_property_map<N, Tuple>:: key_type

typedef to *Tuple**

Nth_of_tuple_property_map<N, Tuple>:: value_type

typedef to *boost::tuples::element<N, Tuple>::type*

Nth_of_tuple_property_map<N, Tuple>:: reference

typedef to *value_type&*

Nth_of_tuple_property_map<N, Tuple>:: category

boost::lvalue_property_map_tag

Creation

Nth_of_tuple_property_map<N, Tuple> pmap; Constructor.

Operations

template<class Iter>
reference *pmap*[*Iter tuple*]

Access a property map element.
Template Parameters:
Iter: Type convertible to *key_type*.

Related Functions

Nth_of_tuple_property_map<N, typename value_type_traits<Iter>::type>

make_nth_of_tuple_property_map(Iter)

Free function to create a *Nth_of_tuple_property_map* property map.

Example

See *property_map.cpp* example.

CGAL::Second_of_pair_property_map<Pair>

Class

Definition

Property map that accesses the second item of a *std::pair*.

#include <CGAL/property_map.h>

Parameters

template<typename Pair>
struct *Second_of_pair_property_map*;

Parameters:
Pair: Instance of *std::pair*.

Inherits From

boost::put_get_helper< Pair::second_type& , Second_of_pair_property_map<Pair> >

Is Model for the Concepts

Model of *boost::LvaluePropertyMap* concept.

Types

Second_of_pair_property_map<Pair>:: key_type

typedef to *Pair**

Second_of_pair_property_map<Pair>:: value_type

typedef to *Pair::second_type*

Second_of_pair_property_map<Pair>:: reference

typedef to *value_type&*

Second_of_pair_property_map<Pair>:: category

boost::lvalue_property_map_tag

Creation

Second_of_pair_property_map<Pair> pmap; Constructor.

Operations

template<class Iter>
reference *pmap*[*Iter pair*] Access a property map element.
Template Parameters:
Iter: Type convertible to *key_type*.

Related Functions

Second_of_pair_property_map<typename value_type_traits<Iter>::type>

make_second_of_pair_property_map(Iter)

Free function to create a *Second_of_pair_property_map* property map.

See Also

CGAL::First_of_pair_property_map<Pair> . . . . page 4305

Example

See *property_map.cpp* example.

Chapter 76

Handles, Ranges and Circulators

Olivier Devillers, Lutz Kettner, Sylvain Pion, Michael Seel, and Mariette Yvinec

Contents

76.1 Handles

Most data structures in CGAL use the concept of *Handle* in their user interface to refer to the elements they store. This concept describes what is sometimes called a trivial iterator. A *Handle* is akeen to a pointer to an object providing the dereference operator *operator*()* and member access *operator->()* but no increment or decrement operators like iterators. A *Handle* is intended to be used whenever the referenced object is not part of a logical sequence.

Model for a handle A simple pointer *T**, an iterator or a circulator with value type *T*, are also handles.

76.2 Ranges

Most data structures in CGAL use the concept of an iterator range. The *Range* and *ConstRange* concepts encapsulate the access to the first and the past-the-end iterators of an iterator range. STL containers are models of *Range*. The Boost.Range library provides good support around this concept as well.

76.3 Circulators

An introduction to the concept of circulators is given here. A couple of adaptors are presented that convert between iterators and circulators. Some useful functions for circulators follow. This chapter concludes with a discussion of the design decisions taken. For the full description of the circulator requirements, the provided base classes, the circulator tags, and the support for generic algorithms that work for iterators as well as for circulators please refer to the reference pages. Note that circulators are not part of STL, but of CGAL.

76.3.1 Introduction

The concept of iterators in STL is tailored for linear sequences [C++98, MS96]. In contrast, circular sequences occur naturally in many combinatorial and geometric structures. Examples are polyhedral surfaces and planar maps, where the edges emanating from a vertex or the edges around a facet form a circular sequence.

Since circular sequences do not allow for efficient iterators, we have introduced the new concept of *circulators*. They share most of the requirements of iterators, while the main difference is the lack of a past-the-end position in the sequence. Appropriate adaptors are provided between iterators and circulators to integrate circulators smoothly into the framework of STL. An example of a generic `contains` function illustrates the use of circulators. As usual for circular structures, a `do`-`while` loop is preferable, such that for the specific input, `c == d`, all elements in the sequence are reached.

```
template <class Circulator, class T>
bool contains( Circulator c, Circulator d, const T& value) {
    if (c != 0) {
        do {
            if (*c == value)
                return true;
        } while (++c != d);
    }
    return false;
}
```

Three circulator categories are defined: forward, bidirectional and random-access circulators. Given a circulator `c`, the operation `*c` denotes the item the circulator refers to. The operation `++c` advances the circulator by one item and `--c` steps a bidirectional circulator one item backwards. For random-access circulators `c+n` advances the circulator `n` steps. Two circulators can be compared for equality.

Circulators have a different notion of reachability and ranges than iterators. A circulator `d` is called *reachable* from a circulator `c` if `c` can be made equal to `d` with finitely many applications of the operator `++`. Due to the circularity of the sequence this is always true if both circulators refer to items of the same sequence. In particular, `c` is always reachable from `c`. Given two circulators `c` and `d`, the range `[c,d)` denotes all circulators obtained by starting with `c` and advancing `c` until `d` is reached, but does not include `d`, for `d` $\neq$ `c`. So far it is the same range definition as for iterators. The difference lies in the use of `[c,c)` to denote all items in the circular sequence, whereas for an iterator `i` the range `[i,i)` denotes the empty range. As long as `c != d` the range `[c,d)` behaves like an iterator range and could be used in STL algorithms. For circulators however, an additional test `c == NULL` is required that returns true if and only if the circular sequence is empty. As for C++, we recommend the use of 0 instead of `NULL`.

Besides the conceptual cleanness, the main reason for inventing a new concept with a similar intent as iterators is efficiency. An iterator is supposed to be a light-weight object – merely a pointer and a single indirection to advance the iterator. Although iterators could be written for circular sequences, we do not know of an efficient solution. The missing past-the-end situation in circular sequences can be solved with an arbitrary sentinel in the cyclic order, but this would destroy the natural symmetry in the structure (which is in itself a bad idea) and additional bookkeeping in the items and checking in the iterator advance method reduces efficiency. Another solution may use more bookkeeping in the iterator, e.g. with a start item, a current item, and a kind of winding-number that is zero for the `begin()`-iterator and one for the past-the-end situation[1]. We have introduced the concept of circulators that allows light-weight implementations and the CGAL support library provides adaptor classes that convert between iterators and circulators (with the corresponding penalty in efficiency), so as to integrate this new concept into the framework of STL.

A serious design problem is the slight change of the semantic for circulator ranges as compared to iterator ranges. Since this semantic is defined by the intuitive operators `++` and `==`, which we would like to keep for circulators as well, circulator ranges can be used in STL algorithms. This is in itself a useful feature, if there would not be the definition of a full range $[c, c)$ that an STL algorithm will treat as an empty range. However, the likelihood of a mistake may be overestimated, since for a container `C` supporting circulators there is no `end()` member function, and an expression such as `std::sort( C.begin(), C.end())` will fail. It is easy to distinguish iterators and circulators at compile time, which allows for generic algorithms supporting both as arguments. It is also possible to protect algorithms against inappropriate arguments using the same technique, see the reference pages for circulators, specifically the *Assert_iterator* and *is_empty_range* functions.

Warning: Please note that the definition of a range is different from that of iterators. An interface of a data structure must declare whether it works with iterators, circulators, or both. STL algorithms always specify only iterators in their interfaces. A range $[c, d)$ of circulators used in an interface for iterators will work as expected as long as $c \mathrel{!}= d$. A range $[c, c)$ will be interpreted as the empty range like for iterators, which is different than the full range that it should denote for circulators.

76.3.2 Forward Circulator

A class *Circulator* that satisfies the requirements of a forward circulator with the value type T, supports the following operations. See the reference pages for the full set of requirements. Note that the stated return values are not required, only a return value that is convertible to the stated type is required. As for C++, we recommend the use of 0 instead of `NULL`.

Types

Circulator:: value_type	the value type T.
Circulator:: reference	either reference or const reference to T.
Circulator:: pointer	either pointer or const pointer to T.
Circulator:: size_type	unsigned integral type that can hold the size of the sequence.
Circulator:: difference_type	signed integral type that can hold the distance between two circulators.
Circulator:: iterator_category	circulator category *Forward_circulator_tag*.

Creation

Circulator c;	a circulator equal to *NULL* denoting an empty sequence.
Circulator c(d);	a circulator equal to d.

[1] This is currently implemented as the adaptor class which provides a pair of iterators for a given circulator.

Operations

Circulator&	$c = d$	Assignment.
bool	$c == NULL$	Test for emptiness.
bool	$c \mathrel{!}= NULL$	Test for non-emptiness, i.e. *!(c == NULL)*.
bool	$c == d$	*c* is equal to *d* if they refer to the same item.
bool	$c \mathrel{!}= d$	Test for inequality, i.e. *!(c == d)*.
reference	$*c$	Returns the value of the circulator. If *Circulator* is mutable **c = t* is valid. *Precondition*: *c* is dereferenceable.
pointer	$c->$	Returns a pointer to the value of the circulator. *Precondition*: *c* is dereferenceable.
Circulator&	$++c$	Prefix increment operation. *Precondition*: *c* is dereferenceable. *Postcondition*: *c* is dereferenceable.
Circulator	$c++$	Postfix increment operation. The result is the same as that of: *Circulator tmp = c; ++c; return tmp;* .

76.3.3 Bidirectional Circulator

A class *Circulator* that satisfies the requirements of a bidirectional circulator with the value type *T*, supports the following operations in addition to the operations supported by a forward circulator.

Types

Circulator:: iterator_category

circulator category *Bidirectional_circulator_tag*.

Operations

Circulator&	$--c$	Prefix decrement operation. *Precondition*: *c* is dereferenceable. *Postcondition*: *c* is dereferenceable.
Circulator	$c--$	Postfix decrement operation. The result is the same as that of *Circulator tmp = c; --c; return tmp;* .

76.3.4 Random Access Circulator

A class *Circulator* that satisfies the requirements of a random access Circulator for the value type *T*, supports the following operations in addition to the operations supported by a bidirectional Circulator. In contrast to random access iterators, no comparison operators are available for random access circulators.

Types

Circulator:: iterator_category

circulator category *Random_access_circulator_tag*.

Operations

Circulator& $c += difference\_type\ n$

The result is the same as if the prefix increment operation was applied n times, but it is computed in constant time.

Circulator $c + difference\_type\ n$ Same as above, but returns a new circulator.

Circulator $difference\_type\ n + c$ Same as above.

Circulator& $c -= difference\_type\ n$

The result is the same as if the prefix decrement operation was applied n times, but it is computed in constant time.

Circulator $c - difference\_type\ n$ Same as above, but returns a new circulator.

reference $c[\ difference\_type\ n]$ Returns **(c + n)*.

difference_type $c - Circulator\ d$ returns the difference between the two circulators. The value will be in the interval $[1-s, s-1]$ if s is the size of the total sequence. The difference for a fixed circulator c (or d) with all other circulators d (or c) is a consistent ordering of the elements in the data structure. There has to be a minimal circulator $d_{\min}$ for which the difference $c - d_{\min}$ to all other circulators c is non-negative.

Circulator *c.min_circulator() const*

Returns the minimal circulator $c_{\min}$ in constant time.

76.3.5 Adaptors Between Iterators and Circulators

Algorithms working on iterator ranges can not be applied to circulator ranges in full generality, only to subranges (see the warning in Section 76.3.1). The following adaptors convert circulators to iterators and vice versa (with the unavoidable space and time penalty) to reestablish this generality.

#include <CGAL/circulator.h>

Container_from_circulator	container-like class with iterators built from a circulator
Circulator_from_iterator	circulator over a range of two iterators
Circulator_from_container	circulator for a container

The following example applies the generic `std::reverse()` algorithm from STL to a sequence given by a bidirectional circulator `c`. It uses the *Container_from_circulator* adaptor.

```
Circulator c;  // c must be at least bidirectional.
CGAL::Container_from_circulator<Circulator> container(c);
std::reverse( container.begin(), container.end());
```

Another example defines a circulator *c* for a vector of *int*'s. However, since there are no elements in the vector, the circulator denotes an empty sequence. If there were elements in the vector, the circulator would implement a random access modulus the size of the sequence.

```
std::vector<int> v;
typedef  CGAL::Circulator_from_iterator<
             std::vector<int>::iterator > Circulator;
Circulator c( v.begin(), v.end());
```

76.3.6 Functions on Circulators

A few functions deal with circulators and circulator ranges. The type *C* denotes a circulator. The type *IC* denotes either a circulator or an iterator. More on algorithms that work with circulators as well with iterators can be found in the reference pages.

#include <CGAL/circulator.h>

circulator_size(C c)	size of the sequence reachable by *c*
circulator_distance(C c, C d)	number of elements in the range $[c,d)$
iterator_distance(IC ic1, IC ic2)	number of elements in the range $[ic2,ic1)$
is_empty_range(IC ic1, IC ic2)	test the range $[ic2,ic1)$ for emptiness

Handles, Ranges and Circulators Reference Manual

Olivier Devillers, Lutz Kettner, Sylvain Pion, Michael Seel, and Mariette Yvinec

The concept of iterators in the STL is tailored for linear sequences.

CGAL extends this in several directions. First, it supports the notion of *Handle* (also sometimes refered to as the trivial iterator) which is used to document that no traversal operation is needed, only reference to an element. It also uses the *Range* and *ConstRange* concepts which encapsulates the access to both the first and the past-the-end iterators of an iterator range.

Besides linear sequences, circular sequences also occur naturally in many combinatorial and geometric structures. Examples are polyhedral surfaces and planar maps, where the edges emanating from a vertex or the edges around a facet form a circular sequence.

We provide several functions, classes and macros to assist in working with circulators: distance computation, adaptor classes converting between circulators and iterators, base classes to ease the implementation of circulators, and support for generic algorithms that work with circulators as well as with iterators.

76.4 Classified Reference Pages

Concepts

Classes

Functions

Macros

76.5 Alphabetical List of Reference Pages

Function

CGAL::Assert_circulator

Definition

Each of the following assertions, applicable to an iterator or a circulator or both, checks at compile time if its argument is of the kind stated in the assertions name, i.e. a circulator, an iterator, or a particular category of either an iterator or a circulator. Note that neither input nor output circulators exists.

#include <CGAL/circulator.h>

void	*Assert_circulator(C c)*
void	*Assert_iterator(I i)*
void	*Assert_circulator_or_iterator(IC i)*

void	*Assert_input_category(I i)*
void	*Assert_output_category(I i)*
void	*Assert_forward_category(IC ic)*
void	*Assert_bidirectional_category(IC ic)*
void	*Assert_random_access_category(IC ic)*

See Also

Circulator_tag, *Circulator_traits*, *query_circulator_or_iterator*, *Circulator*.

CGAL::circulator_distance

Function

Definition

The distance of a circulator c to a circulator d is the number of elements in the range $[c,d)$. It is defined to be zero for a circulator on an empty sequence and it returns the size of the data structure when applied to a range of the form $[c,c)$.

#include <CGAL/circulator.h>

template <class C>
C::difference_type *circulator_distance(C c, C d)*

See Also

circulator_size, *iterator_distance*, *is_empty_range*, *Circulator*.

CGAL::circulator_size

Definition

The size of a circulator is the size of the data structure it refers to. It is zero for a circulator on an empty sequence. The size can be computed in linear time for forward and bidirectional circulators, and in constant time for random access circulators using the minimal circulator. The function *circulator_size(c)* returns the circulator size. It uses the *c.min_circulator()* function if *c* is a random access circulator.

#include <CGAL/circulator.h>

template <class C>
C::size_type *circulator_size(C c)*

See Also

circulator_distance, *iterator_distance*, *is_empty_range*, *Circulator*.

Circulator

Definition

Note: This specification is a revised version based on the C++ Standard [C++98], which is available now. In particular, iterator traits are now assumed and required.

Iterators in the STL were tailored for linear sequences. The specialization for circular data structures leads to slightly different requirements which we will summarize in the *circulators* concept. The main difference is that a circular data structure has no natural past-the-end value. As a consequence, a container supporting circulators will not have an `end()`-member function. The semantic of a circulator range differs from the semantic of an iterator range. For a circulator c the range $[c,c)$ denotes the sequence of all elements in the data structure. For iterators, this range defines the empty sequence. A separate test for an empty sequence has been added to the circulator requirements: A comparison $c ==$ `NULL` for a circulator c is true for an empty sequence. As for C++, we recommend the use of 0 instead of `NULL`.

Similar to STL iterators, we distinguish between forward, bidirectional, and random access circulators[2]. Most requirements for circulators are equal to those for iterators. We present the changes, please refer to [MS96, chapter 18] or [C++98] for the iterator requirements.

Past-the-end value: There is no past-the-end value for circulators.

Singular values: There are no singular values for circulators[3]

Empty sequence: The comparison $c ==$ `NULL` (or $c == 0$) for a circulator c is `true` if c denotes an empty sequence, and `false` otherwise.

Dereferenceable values: A circulator that does not denote an empty sequence is dereferenceable.

Reachability: Each dereferenceable circulator can reach itself with a finite and non-empty sequence of applications of *operator++*.

Ranges: For any circulator c the range $[c,c)$ is a valid range. If the circulator refers to an empty sequence, the range $[c,c)$ denotes the empty range. Otherwise the circulator is dereferenceable and the range $[c,c)$ denotes the sequence of all elements in the data structure. *Remark:* When a circulator is used in a place of an iterator, as, for example, with an STL algorithm, it will work as expected with the only exception that, in STL algorithms, the range $[c,c)$ denotes always the empty range. This is not a requirement, but a consequence of the requirements stated here and the fact that the STL requirements for iterator ranges are based on the `operator++` and the `operator==`, which we use for circulators as well. In principle, we face here the difference between a `while` loop and a `do-while` loop.

Types: For a circulator of type C the following local types are required:

`C::value_type`	value type the circulator refers to.
`C::reference`	reference type used for the return type of `C::operator*()`.
`C::pointer`	pointer type used for the return type of `C::operator->()`.
`C::size_type`	unsigned integral type that can hold the size of a sequence
`C::difference_type`	signed integral type that can hold the distance between two circulators.
`C::iterator_category`	circulator category.

[2] Input circulators are a contradiction, since any circulator is supposed to return once to itself. Output circulators are not supported since they would be indistinguishable from output iterators.

[3] Since circulators must be implemented as classes anyway, there is no need to allow singular values for them. An un-initalized circulator does not have a singular value, but is supposed to refer to an empty sequence.

Forward Circulators

In the following, we assume that `a` and `b` are circulators of type `C`, `r` is of type `C&` (is assignable), and `T` denotes the value type of `C`. Let `D` be the distance type of `C`. As for C++, we recommend the use of 0 instead of `NULL`.

`C()`	a circulator equal to *NULL* denoting an empty sequence.
`a == NULL`	Returns `true` if `a` denotes an empty sequence, `false` otherwise. For simplicity, `NULL == a` is not required. The behavior for comparisons with pointer-like values different than `NULL` is undefined. A runtime assertion is recommended.
`a != NULL`	Returns `!(a == NULL)`.
`++r`	Like for forward iterators, but a dereferenceable circulator `r` will always be dereferenceable after `++r` (no past-the-end value). *Precondition:* `r` does not denote an empty sequence.
`r++`	Same as for `++r`.
`C::iterator_category`	circulator category `CBP_Forward_circulator_tag`.

Bidirectional Circulators

The same requirements as for the forward circulators hold for bidirectional iterators with the following change of the iterator category:

`C::iterator_category`	circulator category `CBP_Bidirectional_circulator_tag`.

Random Access Circulators

The same requirements as for the bidirectional circulators hold for random access iterators with the following changes and extensions.

The idea of random access extends naturally to circulators using equivalence classes modulus the length of the sequence. With this in mind, the additional requirements for random access iterators hold also for random access circulators. The only exception is that the random access iterator is required to provide a total order on the sequence, which a circulator cannot provide[4].

The difference of two circulators is not unique as for iterators. A reasonable requirement demands that the result is in a certain range $[1 - \mathit{size}, \mathit{size} - 1]$, where *size* is the size of the sequence, and that whenever a circulator `a` is fixed that the differences with all other circulators of the sequence form a consistent ordering.

For the adaptor to iterators a minimal circulator $d_{\min}$ is required for which the difference $c - d_{\min}$ to all other circulators c is non negative.

`b - a`	limited range and consistent ordering as explained above.
`a.min_circulator()`	returns the minimal circulator from the range $[a, a)$.
`C::iterator_category`	circulator category `CBP_Random_access_circulator_tag`.

[4] One might define an order by splitting the circle at a fixed point, e.g. the start circulator provided from the data structure. This is what the adaptor to iterators will do. Nonetheless, we do not require this for circulators.

Const Circulators

As with iterators, we distinguish between circulators and const circulators. The expression `*a = t` with `t` of type `T` is valid for mutable circulators. It is invalid for const circulators.

Circulators in Container Classes

For a container `x` of type `X` that supports circulators `c` the following naming convention is recommended:

`X::Circulator`	the type of the mutable circulator.
`X::Const_circulator`	the type of the const circulator.
`c = x.begin()`	the start circulator of the sequence. It is of type `X::Circulator` for a mutable container or `X::Const_circulator` for a const container. There must not be an `end()` member function.

If a container will support iterators and circulators, the member function `circulator_begin()` is proposed. However, the support of iterators and circulators simultaneously is not recommended, since it would lead to fat interfaces. The natural choice should be supported, the other concept will be available through adaptors.

Example

A generic `contains` function accepts a range of circulators and a value. It returns *true* if the value is contained in the sequence of items denoted by the range of circulators. As usual for circular structures, a `do-while` loop is preferable, such that for the specific input, `c == d`, all elements in the sequence are reached. Note that the example simplifies if the sequence is known to be non-empty, which is for example the common case in polyhedral surfaces where vertices and facets have at least one incident edge.

```
template <class Circulator, class T>
bool contains( Circulator c, Circulator d, const T& value) {
    if (c != 0) {
        do {
            if (*c == value)
                return true;
        } while (++c != d);
    }
    return false;
}
```

CGAL::Circulator_from_container<C>

Definition

The adaptor *Circulator_from_container<C>* provides a circulator for an STL container *C* of equal category as the iterator provided by the container. The iterator must be at least of the forward iterator category. The corresponding non-mutable circulator is called *Const_circulator_from_container<C>*.

The container type *C* is supposed to conform to the STL requirements for container (i.e. to have a *begin()* and an *end()* iterator as well as the local types *reference*, *const_reference*, *value_type*, *size_type*, and *difference_type*).

#include <CGAL/circulator.h>

Types

All types required for circulators are provided.

Creation

Circulator_from_container<C> c; — a circulator *c* on an empty sequence.

Circulator_from_container<C> c(C container);*

a circulator *c* initialized to refer to the first element in *container*, i.e. *container.begin()*. The circulator *c* refers to an empty sequence if the *container* is empty.

Circulator_from_container<C> c(C container, C::iterator i);*

a circulator *c* initialized to refer to the element **i* in *container*.
Precondition: **i* is dereferenceable and refers to *container*.

Operations

The adaptor conforms to the requirements of the corresponding circulator category. An additional member function *current_iterator()* returns the current iterator pointing to the same position as the circulator does.

See Also

Container_from_circulator, *Circulator_from_iterator*, *Circulator*.

Example

The following program composes two adaptors – from a container to a circulator and back to an iterator. It applies an STL sort algorithm on a STL vector with three elements. The resulting vector will be `[2 5 9]` as it is checked by the assertions. The program is part of the CGAL distribution.

```
#include <cassert>
#include <vector>
#include <algorithm>
#include <CGAL/circulator.h>

typedef CGAL::Circulator_from_container< std::vector<int> >  Circulator;
typedef CGAL::Container_from_circulator<Circulator>          Container;
typedef Container::iterator                                   Iterator;

int main() {
    std::vector<int> v;
    v.push_back(5);
    v.push_back(2);
    v.push_back(9);
    Circulator c( &v);
    Container  container( c);
    std::sort( container.begin(), container.end());
    Iterator i = container.begin();
    assert( *i == 2);
    i++;    assert( *i == 5);
    i++;    assert( *i == 9);
    i++;    assert(  i == container.end());
    return 0;
}
```

File: examples/Circulator/circulator_prog2.cpp

Class

CGAL::Circulator_from_iterator<I>

Definition

The adaptor *Circulator_from_iterator<I>* converts two iterators of type *I*, a begin and a past-the-end value, to a circulator of equal category. The iterator must be at least of the forward iterator category. The circulator will be mutable or non-mutable according to the iterator. Iterators provide no *size_type*. This adapter assumes *std::size_t* instead.

#include <CGAL/circulator.h>

Types

typedef I *iterator;*

In addition all types required for circulators are provided.

Creation

Circulator_from_iterator<I> c;

a circulator *c* on an empty sequence.

Circulator_from_iterator<I> c(I begin, I end, I cur = begin);

a circulator *c* initialized to refer to the element **cur* in a range `[`*begin*`,`*end*`)`. The circulator *c* refers to a empty sequence if *begin==end*.

Circulator_from_iterator<I> c(Circulator_from_iterator<I,T,Size,Dist> d, I cur);

a copy of circulator *d* referring to the element **cur*. The circulator *c* refers to a empty sequence if *d* does so.

Operations

The adaptor conforms to the requirements of the respective circulator category. An additional member function *current_iterator()* returns the current iterator pointing to the same position as the circulator does.

See Also

Container_from_circulator, *Circulator_from_container*, *Circulator*.

Example

The following program composes two adaptors – from an iterator to a circulator and back to an iterator. It applies an STL sort algorithm on a STL vector containing three elements. The resulting vector will be `[2 5 9]` as it is checked by the assertions. The program is part of the CGAL distribution.

```
#include <CGAL/basic.h>
#include <cassert>
#include <vector>
#include <algorithm>
#include <CGAL/circulator.h>

typedef  std::vector<int>::iterator                       I;
typedef  CGAL::Circulator_from_iterator<I>                Circulator;
typedef  CGAL::Container_from_circulator<Circulator> Container;
typedef  Container::iterator                              Iterator;

int main() {
    std::vector<int> v;
    v.push_back(5);
    v.push_back(2);
    v.push_back(9);
    Circulator c( v.begin(), v.end());
    Container  container( c);
    std::sort( container.begin(), container.end());
    Iterator i = container.begin();
    assert( *i == 2);
    i++;    assert( *i == 5);
    i++;    assert( *i == 9);
    i++;    assert(  i == container.end());
    return 0;
}
```

File: examples/Circulator/circulator_prog1.cpp

Another example usage for this adaptor is a random access circulator over the built-in C arrays. Given an array of type `T*` with a begin pointer `b` and a past-the-end pointer `e` the adaptor *Circulator_from_iterator<T*> c(b,e)* is a random access circulator *c* over this array.

Class

CGAL::Circulator_tag

Definition

Iterators and circulators as well as different categories of circulators can be distinguished with the use of discriminating functions and the following circulator tags. A couple of base classes simplify the task of writing own circulators. They declare the appropriate tags and the local types needed for circulators. To use the tags or base classes only it is sufficient to include:

#include <CGAL/circulator_bases.h>
#include <CGAL/circulator.h>

Compile Time Tags

struct Circulator_tag {};	any circulator.
struct Iterator_tag {};	any iterator.
struct Forward_circulator_tag {};	derived from *forward_iterator_tag*.
struct Bidirectional_circulator_tag {};	derived from *bidirectional_iterator_tag*.
struct Random_access_circulator_tag {};	derived from *random_access_iterator_tag*.

Base Classes

template < class Category, class T, class Dist = std::ptrdiff_t, class Size = std::size_t, class Ptr = T, class Ref = T& >*
struct Circulator_base {};

struct Forward_circulator_base {};
struct Bidirectional_circulator_base {};
struct Random_access_circulator_base {};

See Also

query_circulator_or_iterator, *Circulator_traits*, *Assert_circulator*, *CGAL_For_all*, *is_empty_range*, *Circulator*.

Example

The above declarations can be used to distinguish between iterators and circulators and between different circulator categories. The assertions can be used to protect a templatized algorithm against instantiations that do not fulfill the requirements. The following example program illustrates both.

```
#include <cassert>
#include <list>
#include <CGAL/circulator.h>

template <class C> inline  int foo( C c, std::forward_iterator_tag) {
```

```
    CGAL::Assert_circulator( c);
    CGAL::Assert_forward_category( c);
    return 1;
}
template <class C> inline  int foo( C c,
   std::random_access_iterator_tag) {
    CGAL::Assert_circulator( c);
    CGAL::Assert_random_access_category( c);
    return 2;
}
template <class I> inline  int foo( I i, CGAL::Iterator_tag) {
    CGAL::Assert_iterator( i);
    return 3;
}

template <class C> inline  int foo( C c, CGAL::Circulator_tag) {
    CGAL::Assert_circulator( c);
    typedef std::iterator_traits<C> Traits;
    typedef typename Traits::iterator_category iterator_category;
    return foo( c, iterator_category());
}
template <class IC> inline  int foo( IC ic) {
    typedef CGAL::Circulator_traits<IC> Traits;
    typedef typename Traits::category category;
    return foo( ic, category());
}

int main() {
    typedef CGAL::Forward_circulator_base<int>       F;
    typedef CGAL::Random_access_circulator_base<int> R;
    F f = F();
    R r = R();
    std::list<int> l;
    assert( foo( f)         == 1);
    assert( foo( r)         == 2);
    assert( foo( l.begin()) == 3);
    return 0;
}
```

File: examples/Circulator/circulator_prog3.cpp

Implementation

Since not all current compilers can eliminate the space needed for the compile time tags even when deriving from them, we implement a variant for each base class that contains a protected *void** data member called *_ptr*. Here, the allocated space in the derived classes can be reused.

template <class T, class Dist, class Size> *class Forward_circulator_ptrbase {};*	forward circulator.
template <class T, class Dist, class Size> *class Bidirectional_circulator_ptrbase {};*	bidirectional circulator.

template <class T, class Dist, class Size> *class Random_access_circulator_ptrbase* {}*;*	random access circulator.

CGAL::Circulator_traits<C>

Definition

The circulator traits class distinguishes between circulators and iterators. It defines a local type *category* that is identical to the type *Circulator_tag* if the iterator category of the argument *C* is a circulator category. Otherwise it is identical to the type *Iterator_tag*.

The local type *iterator_category* gives the corresponding iterator category for circulators, i.e. one of *forward_iterator_tag*, *bidirectional_iterator_tag*, or *random_access_iterator_tag*.

The local type *circulator_category* gives the corresponding circulator category for iterators, i.e. one of *Forward_circulator_tag*, *Bidirectional_circulator_tag*, or *Random_access_circulator_tag*.

#include <CGAL/circulator.h>

Types

Circulator_traits<C>:: category	either *Iterator_tag* or *Circulator_tag*.
Circulator_traits<C>:: iterator_category	corresponding iterator category for circulators.
Circulator_traits<C>:: circulator_category	corresponding circulator category for iterator

Example

A generic function *bar* that distinguishes between a call with a circulator range and a call with an iterator range:

```
template <class I>
void bar( I i, I j, CGAL::Iterator_tag) {
    CGAL::Assert_iterator(i);
    // This function is called for iterator ranges [i,j).
}
template <class C>
void bar( C c, C d, CGAL::Circulator_tag) {
    CGAL::Assert_circulator(c);
    // This function is called for circulator ranges [c,d).
}
template <class IC>
void bar( IC i, IC j) {  // calls the correct function
    return bar( i, j, typename CGAL::Circulator_traits<IC>::category());
}
```

CGAL::Container_from_circulator<C>

Definition

The adaptor *Container_from_circulator<C>* is a class that converts any circulator type *C* to a kind of container class, i.e. a class that provides an *iterator* and a *const_iterator* type and two member functions – *begin()* and *end()* – that return the appropriate iterators. By analogy to STL container classes these member functions return a const iterator in the case that the container itself is constant and a mutable iterator otherwise.

#include <CGAL/circulator.h>

Types

typedef C   *Circulator;*
Container_from_circulator<C>:: iterator
Container_from_circulator<C>:: const_iterator
Container_from_circulator<C>:: value_type
Container_from_circulator<C>:: reference
Container_from_circulator<C>:: const_reference
Container_from_circulator<C>:: pointer
Container_from_circulator<C>:: const_pointer
Container_from_circulator<C>:: size_type
Container_from_circulator<C>:: difference_type

Creation

Container_from_circulator<C> container;

any iterator of *container* will have a singular value.

Container_from_circulator<C> container(C c);

any iterator of *container* will have a singular value if the circulator *c* corresponds to an empty sequence.

Operations

iterator	*container.begin()*	the start iterator.
const_iterator	*container.begin() const*	the start const iterator.
iterator	*container.end()*	the past-the-end iterator.
const_iterator	*container.end() const*	the past-the-end const iterator.

The *iterator* and *const_iterator* types are of the appropriate iterator category. In addition to the operations required for their category, they have a member function *current_circulator()* that returns a circulator pointing to the same position as the iterator does.

See Also

Circulator_from_iterator, *Circulator_from_container*, *Circulator*.

Example

The generic `reverse()` algorithm from the STL can be used with an adaptor if at least a bidirectional circulator `c` is given.

```
Circulator c;  // c is assumed to be a bidirectional circulator.
CGAL::Container_from_circulator<Circulator> container(c);
reverse( container.begin(), container.end());
```

Implementation

The iterator adaptor keeps track of the number of rounds a circulator has done around the ring-like data structure (a kind of winding number). It is used to distinguish between the start position and the end position which will be denoted by the same circulator internally. This winding number is zero for the *begin()*-iterator and one for the *end()*-iterator. It is incremented whenever the internal circulator passes the *begin()* position. Two iterators are equal if their internally used circulators and winding numbers are equal. This is more general than necessary since an iterator equal to *end()*-iterator is not supposed to be incremented any more, which is here still possible in a defined manner.

The implementation is different for random access iterators. The random access iterator has to be able to compute the size of the data structure in constant time. This is for example needed if the difference of the past-the-end iterator and the begin iterator is taken, which is exactly the size of the data structure. Therefore, if the circulator is of the random-access category, the adapter chooses the minimal circulator for the internal anchor position. The minimal circulator is part of the random access circulator requirements, see Page 4324. For the random access iterator the adaptor implements a total ordering relation that is currently not required for random access circulators.

CGAL_For_all

Definition

In order to write algorithms that work with iterator ranges as well as with circulator ranges we have to consider the difference of representing an empty range. For iterators this is the range $[i,i)$, while for circulators it would be $c == NULL$, the empty sequence test. The function *is_empty_range* provides the necessary generic test which accepts an iterator range or a circulator range and says whether the range is empty or not.

#include <CGAL/circulator.h>

A macro *CGAL_For_all(i, j)* simplifies the writing of such simple loops as the one in the example of the function *is_empty_range*. *i* and *j* can be either iterators or circulators. The macro loops through the range [*i, j*). It increments *i* until it reaches *j*. The implementation looks like:

CGAL_For_all(i,j) ≡
```
for ( bool _circ_loop_flag = ! ::CGAL::is_empty_range(i,j);
        _circ_loop_flag;
        _circ_loop_flag = ((++i) != (j))
    )
```

Note that the macro behaves like a *for*-loop. It can be used with a single statement or with a statement block. For bidirectional iterators or circulators, a backwards loop macro *CGAL_For_all_backwards(i, j)* exists that decrements *j* until it reaches *i*.

See Also

iterator_distance, *is_empty_range*, *Circulator_tag*, *Circulator_traits*, *Assert_circulator_or_iterator*, *Circulator*.

Handle

Definition

Most data structures in CGAL use the concept of Handle in their user interface to refer to the elements they store. This concept describes what is sometimes called a trivial iterator. A Handle is akin to a pointer to an object providing the dereference operator *operator*()* and member access *operator->()* but no increment or decrement operators like iterators. A Handle is intended to be used whenever the referenced object is not part of a logical sequence.

Like iterators, the handle can be passed as template argument to *std::iterators_traits* in order to extract its *value_type*, the type of the element pointed to. The *iterator_category* is *void*.

Refines

DefaultConstructible, CopyConstructible, Assignable, EqualityComparable

The default constructed object must be unique as far as the equality operator is concerned (this serves the same purpose as NULL for pointers). (Note that this is not a generally supported feature of iterators of standard containers.)

Dereference

value_type&	$*h$	returns the object pointed to.
*value_type**	h −>	returns a pointer to the object pointed to.

Has Models

pointers
const pointers
iterators
circulators

Concept

Range

Definition

CGAL and the STL heavily use the concepts of iterators and iterator ranges to describe linear sequences of elements, and algorithms operating on these.

The Range concept aims at encapsulating an iterator range, by providing access to the first and past-the-end iterators of a range. The advantage is that the syntax for passing ranges is much more concise than passing two arguments separately.

Ranges come in different categories depending on the category of their iterator : mutable or constant (modifiability of the elements pointed to), and forward, bidirectional or random-access. The category can be queried using *std::iterator_traits* and the corresponding iterator type. Note that the concepts *Range* and *ConstRange* do not require anything on the category or the value type of the iterator. It must be precised in the documentation of any model of these concepts. For example, in the case of a vector of points, one would say: *This type is a model of Range concept, its iterator type is random-access and its value type is Point*.

Boost also offers the Boost.Range library which provides good support for ranges.

Finally, let us note that ranges, in general (especially in template context) need to be passed and returned by (const) reference for efficiency. This is a difference with iterators which are typically passed by value.

Refines

ConstRange

Boost's Range concept

Types

Range:: const_iterator	The constant iterator type.
Range:: iterator	The iterator type. It must be convertible to *const_iterator*.
Range:: size_type	An unsigned integral type that can represent the size of a range.

Member functions

const_iterator	*r.begin() const*	returns the const iterator pointing to the first element.
const_iterator	*r.end() const*	returns the past-the-end const iterator.
iterator	*r.begin()*	returns the iterator pointing to the first element.
iterator	*r.end()*	returns the past-the-end iterator.
size_type	*r.size() const*	returns the size of the range.
bool	*r.empty() const*	returns whether the range is empty.

Has Models

STL containers

ConstRange

Definition

A constant iterator range. Refer to the *Range* concept for more details.

Refines

Boost's Range concept

Types

ConstRange:: const_iterator	The constant iterator type.
ConstRange:: size_type	An unsigned integral type that can represent the size of a range.

Member functions

const_iterator	*r.begin() const*	returns the const iterator pointing to the first element.
const_iterator	*r.end() const*	returns the past-the-end const iterator.
size_type	*r.size() const*	returns the size of the range.
bool	*r.empty() const*	returns whether the range is empty.

Has Models

STL containers
boost::iterator_range

See Also

Range

CGAL::is_empty_range

Definition

In order to write algorithms that work with iterator ranges as well as with circulator ranges we have to consider the difference of representing an empty range. For iterators this is the range $[i,i)$, while for circulators it would be *c == NULL*, the empty sequence test. The function *is_empty_range* provides the necessary generic test which accepts an iterator range or a circulator range and says whether the range is empty or not.

#include <CGAL/circulator.h>

template< class IC>		
bool	*is_empty_range(IC i, IC j)*	is *true* if the range [*i, j*) is empty, *false* otherwise. *Precondition*: *IC* is either a circulator or an iterator type. The range [*i, j*) is valid.

Example

The following function *process_all* accepts a range $[i, j)$ of an iterator or circulator *IC* and processes each element in this range:

```
template <class IC>
void process_all( IC i, IC j) {
    if (! CGAL::is_empty_range( i, j)) {
        do {
            process(*i);
        } while (++i != j);
    }
}
```

See Also

iterator_distance, *CGAL_For_all*, *Circulator_tag*, *Circulator_traits*, *Assert_circulator_or_iterator*, *Circulator*.

Function

CGAL::iterator_distance

Definition

The following function returns the distance between either two iterators or two circulators.

#include <CGAL/circulator.h>

template <class IC>
iterator_traits<IC>::difference_type *iterator_distance(IC ic1, IC ic2)*

See Also

circulator_size, *circulator_distance*, *is_empty_range*, *Circulator_tag*, *Assert_circulator_or_iterator*, *CGAL_For_all*, *Circulator*.

CGAL::query_circulator_or_iterator

Function

Definition

The following function distinguishes between circulators and iterators. It is based on iterator traits [C++98, Mye95] and *Circulator_traits*.

#include <CGAL/circulator.h>

template <class I>
Iterator_tag *query_circulator_or_iterator(I i)*

if the iterator category of *I* belongs to an iterator.

template <class C>
Circulator_tag *query_circulator_or_iterator(C c)*

if the iterator category of *C* belongs to a circulator.

See Also

Circulator_tag, *Circulator_traits*, *Assert_circulator*, *Circulator*.

Chapter 77

Geometric Object Generators

Olivier Devillers, Susan Hert, Michael Hoffmann, Lutz Kettner, and Sven Schönherr

Contents

77.1 Introduction

A variety of generators for geometric objects are provided in CGAL. They are useful as synthetic test data sets, e.g. for testing algorithms on degenerate object sets and for performance analysis.

Two kinds of point generators are provided: first, random point generators and second deterministic point generators. Most random point generators and a few deterministic point generators are provided as input iterators. The input iterators model an infinite sequence of points. The function *CGAL::cpp0x::copy_n()* can be used to copy a finite sequence; see Section 73.9. The iterator adaptor *Counting_iterator* can be used to create finite iterator ranges; see Section 73.9. Other generators are provided as functions that write to output iterators. Further functions add degeneracies or random perturbations.

In 2D, we provide input iterators to generate random points in a disc (*Random_points_in_disc_2*), in a square (*Random_points_in_square_2*), on a circle (*Random_points_on_circle_2*), on a segment (*Random_points_on_segment*), and on a square (*Random_points_on_square_2*). For generating grid points we provide three functions, *points_on_segment_2*, *points_on_square_grid_2* that write to output iterators and an input iterator *Points_on_segment_2*.

For 3D points, input iterators are provided for random points uniformly distributed in a sphere (*Random_points_in_sphere_3*) or cube (*Random_points_in_cube_3*) or on the boundary of a sphere (*Random_points_on_sphere_3*). For generating 3D grid points, we provide the function *points_on_cube_grid_3* that writes to an output iterator.

For higher dimensions, input iterators are provided for random points uniformly distributed in a d-dimensional cube (*Random_points_in_cube_d*) or d-dimensional ball (*Random_points_in_ball_d*) or on the boundary of a sphere (*Random_points_on_sphere_d*). For generating grid points, we provide the function *points_on_grid_d* that writes to an output iterator.

We also provide two functions for generating more complex geometric objects. The function *random_convex_set_2* computes a random convex planar point set of a given size where the points are drawn from a specific domain and *random_polygon_2* generates a random simple polygon from points drawn from a specific domain.

77.1.1 Random Perturbations

Degenerate input sets like grid points can be randomly perturbed by a small amount to produce *quasi*-degenerate test sets. This challenges numerical stability of algorithms using inexact arithmetic and exact predicates to compute the sign of expressions slightly off from zero. For this the function *perturb_points_2* is provided.

77.1.2 Adding Degeneracies

For a given point set certain kinds of degeneracies can be produced by adding new points. The *random_selection()* function is useful for generating multiple copies of identical points. The function *random_collinear_points_2()* adds collinearities to a point set.

77.1.3 Support Functions and Classes for Generators

The function *random_selection* chooses n items at random from a random access iterator range which is useful to produce degenerate input data sets with multiple entries of identical items.

77.2 Example Generating Degenerate Point Sets

We want to generate a test set of 1000 points, where 60% are chosen randomly in a small disc, 20% are from a larger grid, 10% are duplicates points, and 10% collinear points. A random shuffle removes the construction order from the test set. See Figure 77.1 for the example output.

```
#include <CGAL/Simple_cartesian.h>
#include <cassert>
#include <vector>
#include <algorithm>
#include <CGAL/point_generators_2.h>
#include <CGAL/algorithm.h>
#include <CGAL/random_selection.h>

using namespace CGAL;

typedef Simple_cartesian<double>          R;
```

```
typedef R::Point_2                          Point;
typedef Creator_uniform_2<double,Point>  Creator;
typedef std::vector<Point>                  Vector;

int main() {
    // Create test point set. Prepare a vector for 1000 points.
    Vector points;
    points.reserve(1000);

    // Create 600 points within a disc of radius 150.
    Random_points_in_disc_2<Point,Creator> g( 150.0);
    CGAL::cpp0x::copy_n( g, 600, std::back_inserter(points));

    // Create 200 points from a 15 x 15 grid.
    points_on_square_grid_2( 250.0, 200,
   std::back_inserter(points),Creator());

    // Select 100 points randomly and append them at the end of
    // the current vector of points.
    random_selection( points.begin(), points.end(), 100,
                      std::back_inserter(points));

    // Create 100 points that are collinear to two randomly chosen
    // points and append them to the current vector of points.
    random_collinear_points_2( points.begin(), points.end(), 100,
                               std::back_inserter( points));

    // Check that we have really created 1000 points.
    assert( points.size() == 1000);

    // Use a random permutation to hide the creation history
    // of the point set.
    std::random_shuffle( points.begin(), points.end(), default_random);

    // Check range of values.
    for ( Vector::iterator i = points.begin(); i != points.end(); i++){
        assert( i->x() <=  251);
        assert( i->x() >= -251);
        assert( i->y() <=  251);
        assert( i->y() >= -251);
    }
    return 0;
}
```

File: examples/Generator/random_degenerate_point_set.cpp

77.3 Example Generating Grid Points

The second example demonstrates the point generators with integer points. Arithmetic with *double*s is sufficient to produce regular integer grids. See Figure 77.2 for the example output.

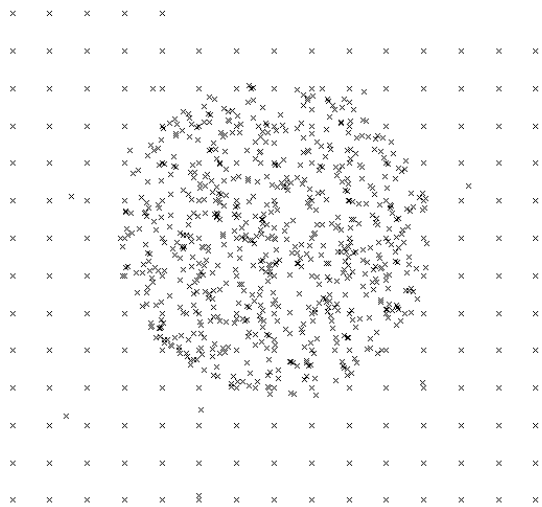


Figure 77.1: Output of example program for point generators.

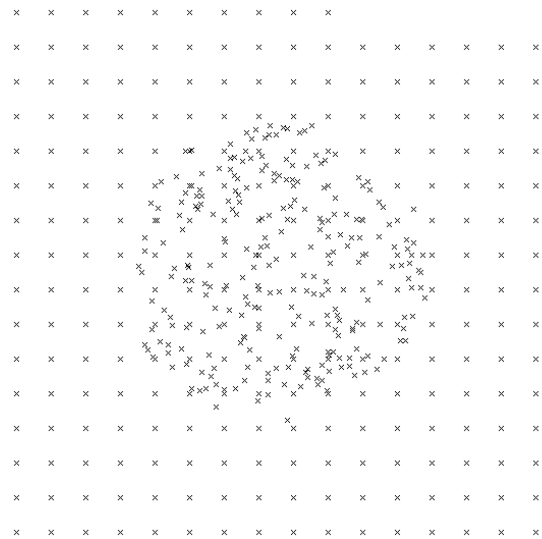


Figure 77.2: Output of example program for point generators working on integer points.

```
#include <CGAL/Simple_cartesian.h>
#include <cassert>
#include <vector>
#include <algorithm>
#include <CGAL/point_generators_2.h>
#include <CGAL/algorithm.h>

using namespace CGAL;

typedef Simple_cartesian<int>          K;
typedef K::Point_2                     Point;
typedef Creator_uniform_2<int,Point>   Creator;

int main() {
    // Create test point set. Prepare a vector for 400 points.
    std::vector<Point> points;
    points.reserve(400);

    // Create 250 points from a 16 x 16 grid. Note that the double
    // arithmetic _is_ sufficient to produce exact integer grid points.
    // The distance between neighbors is 34 pixel = 510 / 15.
    points_on_square_grid_2( 255.0, 250,
   std::back_inserter(points),Creator());

    // Lower, left corner.
    assert( points[0].x() == -255);
    assert( points[0].y() == -255);

    // Upper, right corner. Note that 6 points are missing to fill the
   grid.
    assert( points[249].x() == 255 - 6 * 34);
    assert( points[249].y() == 255);
```

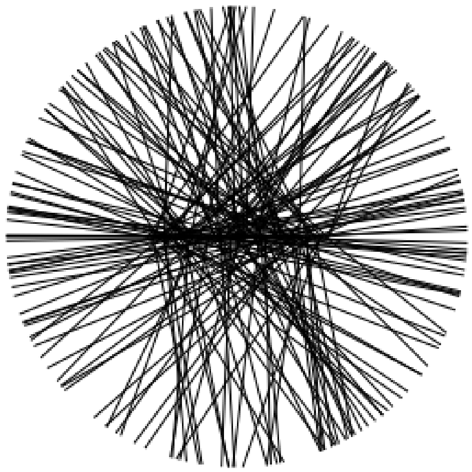


Figure 77.3: Output of the first example program for the generic generator.

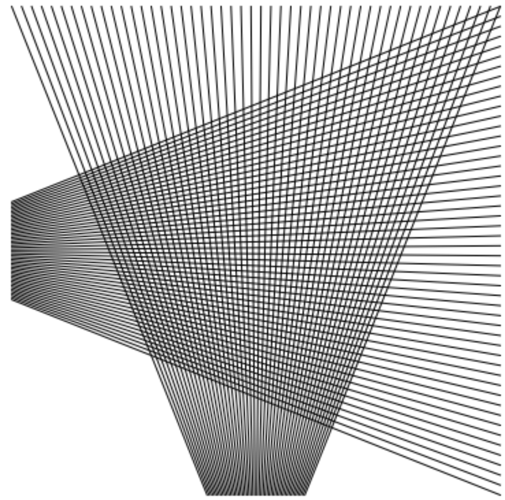


Figure 77.4: Output of the second example program for the generic generator without using intermediate storage.

```
    // Create 250 points within a disc of radius 150.
    Random_points_in_disc_2<Point,Creator> g( 150.0);
    CGAL::cpp0x::copy_n( g, 250, std::back_inserter(points));

    // Check that we have really created 500 points.
    assert( points.size() == 500);
    return 0;
}
```

File: examples/Generator/random_grid.cpp

77.4 Examples Generating Segments

The following two examples illustrate the use of the generic functions from Section 73.9 like *Join_input_iterator_2* to generate composed objects from other generators – here two-dimensional segments from two point generators.

We want to generate a test set of 200 segments, where one endpoint is chosen randomly from a horizontal segment of length 200, and the other endpoint is chosen randomly from a circle of radius 250. See Figure 77.3 for the example output.

```
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <cassert>
#include <vector>
#include <algorithm>
#include <CGAL/Point_2.h>
#include <CGAL/Segment_2.h>
#include <CGAL/point_generators_2.h>
```

```
#include <CGAL/function_objects.h>
#include <CGAL/Join_input_iterator.h>
#include <CGAL/algorithm.h>

using namespace CGAL;

typedef Exact_predicates_inexact_constructions_kernel K;
typedef K::Point_2                               Point;
typedef Creator_uniform_2<double,Point>  Pt_creator;
typedef K::Segment_2                             Segment;
typedef std::vector&lt;Segment&gt;                     Vector;


int main() {
    // Create test segment set. Prepare a vector for 200 segments.
    Vector segs;
    segs.reserve(200);

    // Prepare point generator for the horizontal segment, length 200.
    typedef  Random_points_on_segment_2<Point,Pt_creator>  P1;
    P1 p1( Point(-100,0), Point(100,0));

    // Prepare point generator for random points on circle, radius 250.
    typedef  Random_points_on_circle_2<Point,Pt_creator>  P2;
    P2 p2( 250);

    // Create 200 segments.
    typedef Creator_uniform_2< Point, Segment> Seg_creator;
    typedef Join_input_iterator_2< P1, P2, Seg_creator> Seg_iterator;
    Seg_iterator g( p1, p2);
    CGAL::cpp0x::copy_n( g, 200, std::back_inserter(segs));

    assert( segs.size() == 200);
    for ( Vector::iterator i = segs.begin(); i != segs.end(); i++){
        assert( i->source().x() <=  100);
        assert( i->source().x() >= -100);
        assert( i->source().y() ==    0);
        assert( i->target().x() * i->target().x() +
                i->target().y() * i->target().y() <=  251*251);
        assert( i->target().x() * i->target().x() +
                i->target().y() * i->target().y() >=  249*249);
    }
    return 0;
}
```

File: examples/Generator/random_segments1.cpp

The second example generates a regular structure of 100 segments; see Figure 77.4 for the example output. It uses the *Points_on_segment_2* iterator, *Join_input_iterator_2* and *Counting_iterator* to avoid any intermediate storage of the generated objects until they are used.

```
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
```

```cpp
#include <algorithm>
#include <vector>
#include <CGAL/point_generators_2.h>
#include <CGAL/function_objects.h>
#include <CGAL/Join_input_iterator.h>
#include <CGAL/Counting_iterator.h>

using namespace CGAL;

typedef Exact_predicates_inexact_constructions_kernel K;
typedef K::Point_2                                      Point;
typedef K::Segment_2                                    Segment;
typedef Points_on_segment_2<Point>                      PG;
typedef Creator_uniform_2< Point, Segment>              Creator;
typedef Join_input_iterator_2< PG, PG, Creator>         Segm_iterator;
typedef Counting_iterator<Segm_iterator,Segment>        Count_iterator;
typedef std::vector&lt;Segment&gt;                            Vector;

int main() {
    // Create test segment set. Prepare a vector for 100 segments.
    Vector segs;
    segs.reserve(100);

    // A horizontal like fan.
    PG p1( Point(-250, -50), Point(-250, 50),50);   // Point generator.
    PG p2( Point( 250,-250), Point( 250,250),50);
    Segm_iterator  t1( p1, p2);                     // Segment generator.
    Count_iterator t1_begin( t1);                   // Finite range.
    Count_iterator t1_end( t1, 50);
    std::copy( t1_begin, t1_end, std::back_inserter(segs));

    // A vertical like fan.
    PG p3( Point( -50,-250), Point(  50,-250),50);
    PG p4( Point(-250, 250), Point( 250, 250),50);
    Segm_iterator  t2( p3, p4);
    Count_iterator t2_begin( t2);
    Count_iterator t2_end( t2, 50);
    std::copy( t2_begin, t2_end, std::back_inserter(segs));

    CGAL_assertion( segs.size() == 100);
    for ( Vector::iterator i = segs.begin(); i != segs.end(); i++){
        CGAL_assertion( i->source().x() <=  250);
        CGAL_assertion( i->source().x() >= -250);
        CGAL_assertion( i->source().y() <=  250);
        CGAL_assertion( i->source().y() >= -250);
        CGAL_assertion( i->target().x() <=  250);
        CGAL_assertion( i->target().x() >= -250);
        CGAL_assertion( i->target().y() <=  250);
        CGAL_assertion( i->target().y() >= -250);
    }
    return 0;
}
```

File: examples/Generator/random_segments2.cpp

77.5 Example Generating Point Sets in d dimensions

The following example generates points inside a cube in dimension 5 (examples for ball and sphere are available in the example directory) :

```
#include <iostream>
#include <vector>
#include <CGAL/Cartesian_d.h>
#include <CGAL/point_generators_d.h>

typedef CGAL::Cartesian_d<double>                          Kd;
typedef Kd::Point_d                                        Point;

int main ()
{
  int nb_points = 10;
  int dim =5;
  double size = 100.0;
  std::cout << "Generating "<<nb_points<<" random points in a cube in "
        <<dim<<"D, coordinates from "<<-size<<" to "<<size<<std::endl;
  std::vector<Point> v;
  v.reserve (nb_points);
  CGAL::Random_points_in_cube_d<Point> gen (dim, size);
  for (int i = 0; i < nb_points; ++i) v.push_back (*gen++);
  for (int i = 0; i < nb_points; ++i) std::cout<<" "<<v[i]<<std::endl;
  return 0;
}
```

File: examples/Generator/cube_d.cpp

The output of this example looks like:

```
Generating 10 random points in a cube in 5D, coordinates from -100 to 100
 5 32.9521 26.0403 59.3979 -99.2553 15.5102
 5 80.3731 30.809 7.32491 -90.2544 94.5635
 5 -71.3412 -31.933 -98.0734 79.6493 66.6104
 5 -78.5065 -58.2397 -33.9096 81.2196 57.2512
 5 21.4093 26.7661 57.6083 23.4958 93.1047
 5 10.5895 -21.8914 70.9726 36.756 -42.2667
 5 23.9813 54.4519 -26.0894 -85.18 -21.0775
 5 -48.7499 59.9873 6.22335 -4.16011 81.0727
 5 -11.6615 5.53147 -32.6578 -79.9283 44.5679
 5 53.0183 78.3228 -28.5665 83.3503 68.0482
```

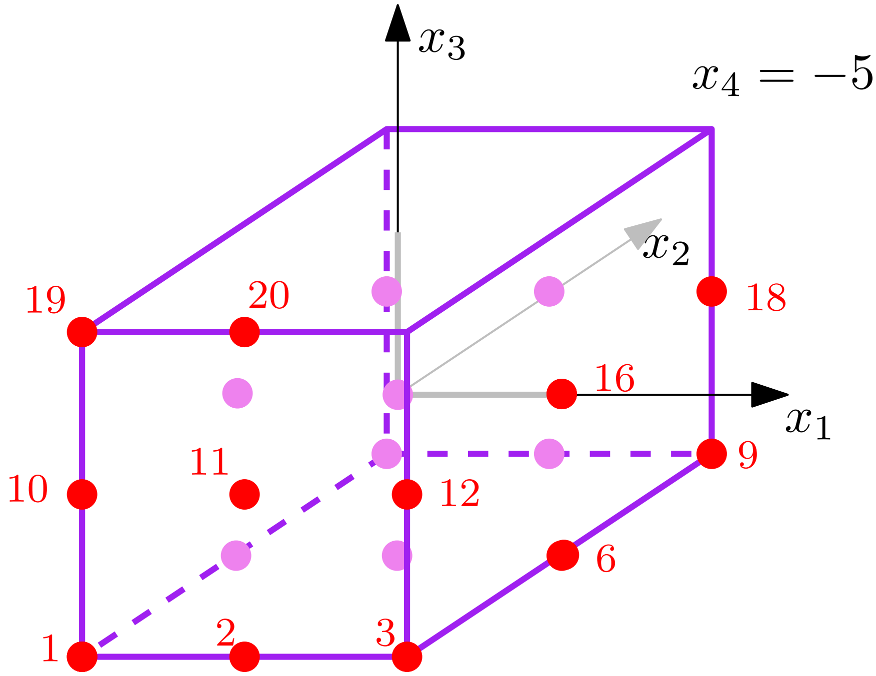
Next example generates grid points in dimension $dim = 4$. Since the required number of points, 20 is between 2^{dim} and 3^{dim} the supporting grid has $3 \times 3 \times 3 \times 3$ points. Since the size parameter is 5, the coordinates are in $\{-5, 0, 5\}$, but since the number of points verifies $20 \le 3^{dim-1}$, all generated points have the same last coordinate -5.

```
#include <iostream>
#include <vector>
#include <CGAL/Cartesian_d.h>
#include <CGAL/point_generators_d.h>
#include <CGAL/constructions_d.h>

typedef CGAL::Cartesian_d<double>                                 Kd;
typedef Kd::Point_d                                               Point;
typedef CGAL::Creator_uniform_d
          <std::vector<double>::iterator, Point>                  Creator_d;

int main ()
{
  int nb_points = 20;
  int dim = 4;
  double size = 5.0;
  std::cout << "Generating "<<nb_points<<" grid points in "
              <<dim<<"D" << std::endl;
  std::vector<Point> v;
  v.reserve(nb_points);
  CGAL::points_on_cube_grid_d (dim, size, (std::size_t) nb_points,
                               std::back_inserter(v), Creator_d(dim) );
  for (int i = 0; i < nb_points; ++i) std::cout<<"  "<<v[i]<<std::endl;
  return 0;
}
```

File: examples/Generator/grid_d.cpp

The output of previous example corresponds to the points of this figure depicted in red or pink (pink points are "inside" the cube). The output is:

```
Generating 20 grid points in 4D
  4 -5 -5 -5 -5
  4 0 -5 -5 -5
  4 5 -5 -5 -5
  4 -5 0 -5 -5
  4 0 0 -5 -5
  4 5 0 -5 -5
  4 -5 5 -5 -5
  4 0 5 -5 -5
  4 5 5 -5 -5
  4 -5 -5 0 -5
  4 0 -5 0 -5
  4 5 -5 0 -5
  4 -5 0 0 -5
  4 0 0 0 -5
  4 5 0 0 -5
  4 -5 5 0 -5
  4 0 5 0 -5
  4 5 5 0 -5
  4 -5 -5 5 -5
  4 0 -5 5 -5
```



77.6 Design and Implementation History

Lutz Kettner coded generators in 2D and 3D For points *in* and *on* sphere, points are generated in a cube up to the moment the point is inside the sphere, then it is normalized to go on the boundary if needed.

Sven Schnherr implemented the Random class.

Michael Hoffmann coded the random convex polygon,

Geert-Jan Giezeman and Susan Hert coded the random simple polygon.

Olivier Devillers coded generators in high dimensions. For points *in ball* and *on sphere*, points are generated on a sphere/ball boundary as a product of normal distributions, then it is normalized. If needed a random radius (with relevant distribution) is used to put the point inside the ball.

Geometric Object Generators Reference Manual

Olivier Devillers, Susan Hert, Michael Hoffmann, Lutz Kettner, and Sven Schönherr

This chapter describes the functions and classes provided in CGAL that are useful for generating synthetic test data sets, *e.g.,* for testing algorithms on degenerate object sets and for performance analysis. These include a class for generating random numbers and function for selecting random items from a set of objects, generators for two-dimensional and three-dimensional points sets, a generator for random convex sets and one for simple polygons. The STL algorithm *std::random_shuffle* is useful with these functions and classes to to achieve random permutations for otherwise regular generators (*e.g.*, points on a grid or segment).

77.7 Classified Reference Pages

Concepts

Functions

Classes

Traits Class

77.8 Alphabetical List of Reference Pages

Variable

CGAL::default_random

Definition

The variable *default_random* is the default random numbers generator used for the generator functions and classes.

#*include* <*CGAL/Random.h*>

Random   *default_random;*

CGAL::perturb_points_2

Function

Definition

The function *perturb_points_2* perturbs each point in a given range of points by a random amount.

#include <CGAL/point_generators_2.h>

template <class ForwardIterator>
void perturb_points_2(
ForwardIterator first,
ForwardIterator last,
double xeps,
double yeps = xeps,
Random& rnd = default_random,
Creator creator = Creator_uniform_2<Kernel_traits<P>::Kernel::RT,P>)

perturbs the points in the range $[first, last)$ by replacing each point with a random point from the *xeps* $\times$ *yeps* rectangle centered at the original point. Two random numbers are needed from *rnd* for each point.

Requirements

- *Creator* must be a function object accepting two *double* values *x* and *y* and returning an initialized point *(x,y)* of type *P*.

  Predefined implementations for these creators like the default are described in Section 73.9.
- The *value_type* of the *ForwardIterator* must be assignable to *P*.
- *P* is equal to the *value_type* of the *ForwardIterator* when using the default initializer.
- The expressions *to_double((*first).x())* and *to_double((*first).y())* must result in the respective coordinate values.

See Also

CGAL::points_on_segment_2 . . . . page 4360
CGAL::points_on_square_grid_2 . . . . page 4363
CGAL::random_selection . . . . page 4373
CGAL::random_selection . . . . page 4373
std::random_shuffle

Function

CGAL::points_on_segment_2

Definition

The function *points_on_segment_2* generates a set of points equally spaced on a segment given the endpoints of the segment.

#include <CGAL/point_generators_2.h>

template <class P, class OutputIterator>
OutputIterator *points_on_segment_2(P p, P q, std::size_t n, OutputIterator o)*

creates n points equally spaced on the segment from p to q, i.e. $\forall i : 0 \leq i < n : o[i] := \frac{n-i-1}{n-1} p + \frac{i}{n-1} q$. Returns the value of o after inserting the n points.

See Also

CGAL::points_on_segment_2 . . . . page 4360
CGAL::points_on_square_grid_2 . . . . page 4363
CGAL::random_collinear_points_2 . . . . page 4367

CGAL::Points_on_segment_2<Point_2>

Class

Definition

The class *Points_on_segment_2<Point_2>* is a generator for points on a segment whose endpoints are specified upon construction. The points are equally spaced

Is Model for the Concepts

PointGenerator . . . . page 4366

Types

typedef std::input_iterator_tag	*iterator_category;*
typedef Point_2	*value_type;*
typedef std::ptrdiff_t	*difference_type;*
*typedef const Point_2**	*pointer;*
typedef Point_2	*reference;*

Creation

Points_on_segment_2<Point_2> g(Point_2 p, Point_2 q, std::size_t n, std::size_t i = 0);

g is an input iterator creating points of type P equally spaced on the segment from p to q. $n-i$ points are placed on the segment defined by p and q. Values of the index parameter i larger than 0 indicate starting points for the sequence further from p. Point p has index value 0 and q has index value $n-1$. *Requirement*: The expressions *to_double(p.x())* and *to_double(p.y())* must result in the respective *double* representation of the coordinates of p, and similarly for q.

Operations

double	*g.range()*	returns the range in which the point coordinates lie, i.e. $\forall x : \|x\| \leq$ *range()* and $\forall y : \|y\| \leq$*range()*

.

Point_2	*g.source()*	returns the source point of the segment.
Point_2	*g.target()*	returns the target point of the segment.

See Also

CGAL::cpp0x::copy_n . . . . page 4174
CGAL::Counting_iterator . . . . page **??**

CGAL::points_on_square_grid_2

Function

Definition

The function *points_on_square_grid_2* generates a given number of points on a square grid whose size is determined by the number of points to be generated.

#include <CGAL/point_generators_2.h>

```
template <class OutputIterator, Creator creator>
OutputIterator          points_on_square_grid_2(
                            double a,
                            std::size_t n,
                            OutputIterator o,
                            Creator creator = Creator_uniform_2<Kernel_traits<P>::Kernel::RT,P>)
```

creates the first n points on the regular $\lceil\sqrt{n}\,\rceil \times \lceil\sqrt{n}\,\rceil$ grid within the square $[-a,a] \times [-a,a]$. Returns the value of o after inserting the n points.

Requirements

- *Creator* must be a function object accepting two *double* values x and y and returning an initialized point *(x,y)* of type P. Predefined implementations for these creators like the default can be found in Section 73.9.
- The *OutputIterator* must accept values of type P. If the *OutputIterator* has a *value_type* the default initializer of the *creator* can be used. P is set to the *value_type* in this case.

See Also

CGAL::perturb_points_2 . . . . page 4359
CGAL::points_on_segment_2 . . . . page 4360
CGAL::points_on_cube_grid_3 . . . . page 4364
CGAL::random_collinear_points_2 . . . . page 4367
CGAL::random_selection . . . . page 4373

std::random_shuffle

Function

CGAL::points_on_cube_grid_3

Definition

The function *points_on_cube_grid_3* generates a given number of points on a cubic grid whose size is determined by the number of points to be generated.

#include <CGAL/point_generators_3.h>

template <class OutputIterator, Creator creator>
OutputIterator *points_on_cube_grid_3(*
double a,
std::size_t n,
OutputIterator o,
Creator creator = Creator_uniform_3<Kernel_traits<P>::Kernel::RT,P>)

creates the first n points on the regular $\lceil n^{1/3} \rceil \times \lceil n^{1/3} \rceil \times \lceil n^{1/3} \rceil$ grid within the cube $[-a,a] \times [-a,a] \times [-a,a]$. Returns the value of o after inserting the n points.

Requirements

- *Creator* must be a function object accepting three *double* values x, y, and z and returning an initialized point *(x,y,z)* of type P. Predefined implementations for these creators like the default can be found in Section 73.9.
- The *OutputIterator* must accept values of type P. If the *OutputIterator* has a *value_type* the default initializer of the *creator* can be used. P is set to the *value_type* in this case.

CGAL::points_on_square_grid_2 . . . . . . page 4363
CGAL::random_selection . . . . . . page 4373

CGAL::points_on_cube_grid_d

Function

Definition

The function *points_on_cube_grid_d* generates a given number of points on a cubic grid in any dimension whose size is determined by the number of points to be generated.

#include <CGAL/point_generators_d.h>

template <class OutputIterator, Creator creator>
OutputIterator *points_on_cube_grid_d(int dim, double a, std::size_t n, OutputIterator o, Creator creator)*

creates the first n points on the regular $\lceil n^{1/dim} \rceil \times \lceil n^{1/dim} \rceil \times \ldots \times \lceil n^{1/dim} \rceil$ grid within the hypercube $[-a,a]^{dim}$. Returns the value of o after inserting the n points.

Requirements

- *Creator* must be a functor accepting an integer (the dimension) and two iterators and returning an initialized point of type P whose coordinates are given by the iterator. For example: *Creator_uniform_d< Kernel_traits<Point_d>::Kernel::RT, Point_d>*. The dimension of *Creator* should be *dim*.
- The *OutputIterator* must accept values of type P.

CGAL::points_on_square_grid_2 . . . . page 4363
CGAL::points_on_cube_grid_3 . . . . page 4364

Concept

PointGenerator

Definition

The concept PointGenerator defines the requirements for a point generator, which can be used in places where input iterators are called for.

Refines

InputIterator

Has Models

Types

PointGenerator:: value_type	the type of point being generated.

Operations

double	*pg.range() const*	return an absolute bound for the coordinates of all generated points.

Function

CGAL::random_collinear_points_2

Definition

#include <CGAL/point_generators_2.h>

template <class RandomAccessIterator, class OutputIterator>
OutputIterator random_collinear_points_2(RandomAccessIterator first, RandomAccessIterator last, std::size_t n, OutputIterator first2, Random& rnd = default_random, Creator creator = Creator_uniform_2<Kernel_traits<P>::Kernel::RT,P>)

randomly chooses two points from the range $[first, last)$, creates a random third point on the segment connecting these two points, writes it to *first2*, and repeats this *n* times, thus writing *n* points to *first2* that are collinear with points in the range $[first, last)$. Three random numbers are needed from *rnd* for each point. Returns the value of *first2* after inserting the *n* points.

Requirements

- *Creator* must be a function object accepting two *double* values *x* and *y* and returning an initialized point *(x,y)* of type *P*. Predefined implementations for these creators like the default can be found in Section 73.9.
- The *value_type* of the *RandomAccessIterator* must be assignable to *P*. *P* is equal to the *value_type* of the *RandomAccessIterator* when using the default initializer.
- The expressions *to_double((*first).x())* and *to_double((*first).y())* must result in the respective coordinate values.

See Also

CGAL::perturb_points_2 . . . . page 4359
CGAL::points_on_segment_2 . . . . page 4360
CGAL::points_on_square_grid_2 . . . . page 4363
CGAL::random_selection . . . . page 4373
std::random_shuffle

Function

CGAL::random_convex_set_2

Definition

The function *random_convex_set_2* computes a random convex planar point set of given size where the points are drawn from a specific domain.

#include <CGAL/random_convex_set_2.h>

template < class OutputIterator, class PointGenerator, class Traits >
OutputIterator random_convex_set_2(std::size_t n, OutputIterator o, PointGenerator pg, Traits t = Default_traits)

computes a random convex n-gon by writing its vertices (oriented counterclockwise) to o. The resulting polygon is scaled such that it fits into the bounding box as specified by pg. Therefore we cannot easily describe the resulting distribution.
Precondition: $n \geq 3$.

Requirements

1. *PointGenerator* is a model of the concept PointGenerator
2. *Traits* is a model of the concept RandomConvexSetTraits_2
3. *Point_generator::value_type* is equivalent to *Traits::Point_2* and *OutputIterator::value_type*.
4. if *Traits* is not specified, *Point_generator::value_type* must be *Point_2*< *R* > for some representation class *R*,

The default traits class *Default_traits* is *Random_convex_set_traits_2*. .

See Also

CGAL::Random_points_in_square_2<Point_2, Creator> . . . . page 4385
CGAL::Random_points_in_disc_2<Point_2, Creator> . . . . page 4383

Implementation

The implementation uses the centroid method described in [Sch96] and has a worst case running time of $O(r \cdot n + n \cdot \log n)$, where r is the time needed by pg to generate a random point.

Example

The following program displays a random convex 500-gon where the points are drawn uniformly from the unit square centered at the origin.

```
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/point_generators_2.h>
#include <CGAL/random_convex_set_2.h>

#include <iostream>
#include <iterator>

typedef CGAL::Exact_predicates_inexact_constructions_kernel K;
typedef K::Point_2                         Point_2;
typedef CGAL::Random_points_in_square_2<
     Point_2,
     CGAL::Creator_uniform_2< double, Point_2 > >
                                           Point_generator;
int main() {
  // create 500-gon and write it into a window:
  CGAL::random_convex_set_2(
            500,
            std::ostream_iterator<Point_2>(std::cout, "\n"),
            Point_generator( 0.5));
  return 0;
}
```

File: examples/Generator/random_convex_set.cpp

Function

CGAL::random_polygon_2

Definition

The function *random_polygon_2* constructs a random simple polygon from points that are drawn from a specific domain. Though each simple polygon defined on this set of points has a non-zero probability of being constructed, some polygons may have higher probabilities than others. The overall distribution of the generated polygons is not known since it depends on the generated points.

#include <CGAL/random_polygon_2.h>

template < class OutputIterator, class PointGenerator, class Traits >
OutputIterator random_polygon_2(std::size_t n, OutputIterator result, PointGenerator pg, Traits t = Default_traits)

computes a random simple polygon by writing its vertices (oriented counterclockwise) to *result*. The polygon generated will have a number of vertices equal to the number of unique points in the first n points generated by *pg*.

Requirements

1. *Traits* is a model of the concept RandomPolygonTraits_2
2. *PointGenerator::value_type* is equivalent to *Traits::Point_2* and *OutputIterator::value_type*.

The default traits class *Default_traits* is the kernel in which *Traits::Point_2* is defined.

See Also

CGAL::Random_points_in_disc_2<Point_2, Creator> . . . . page 4383
CGAL::Random_points_in_square_2<Point_2, Creator> . . . . page 4385

Implementation

The implementation is based on the method of eliminating self-intersections in a polygon by using so-called "2-opt" moves. Such a move eliminates an intersection between two edges by reversing the order of the vertices between the edges. No more than $O(n^3)$ such moves are required to simplify a polygon defined on n points [vLS82]. Intersecting edges are detected using a simple sweep through the vertices and then one intersection is chosen at random to eliminate after each sweep. The worse-case running time is therefore $O(n^4 \log n)$.

Example

The following program displays a random simple polygon with up to 100 vertices, where the vertex coordinates are drawn uniformly from the unit square centered at the origin.

```
#include <CGAL/Simple_cartesian.h>
#include <CGAL/Polygon_2.h>
#include <CGAL/point_generators_2.h>
#include <CGAL/random_polygon_2.h>
#include <CGAL/Random.h>
#include <CGAL/algorithm.h>

#ifdef CGAL_USE_GMP
#include <CGAL/Gmpz.h>
typedef CGAL::Gmpz RT;
#else
// NOTE: the choice of double here for a number type may cause problems
//       for degenerate point sets
#include <CGAL/double.h>
typedef double RT;
#endif


#include <fstream>
#include <list>

typedef CGAL::Simple_cartesian<RT>                        K;
typedef K::Point_2                                        Point_2;
typedef std::list<Point_2>                                Container;
typedef CGAL::Polygon_2<K, Container>                     Polygon_2;
typedef CGAL::Creator_uniform_2<int, Point_2>             Creator;
typedef CGAL::Random_points_in_square_2<Point_2, Creator>
   Point_generator;

const double RADIUS = 100;
const int MAX_POLY_SIZE = 100;

int main( )
{
   Polygon_2            polygon;
   std::list<Point_2>   point_set;
   CGAL::Random         rand;

   int size = rand.get_int(4, MAX_POLY_SIZE);

   // copy size points from the generator, eliminating duplicates, so the
   // polygon will have <= size vertices
   CGAL::copy_n_unique(Point_generator(RADIUS), size,
                       std::back_inserter(point_set));

   std::ostream_iterator< Point_2 >  out( std::cout, " " );
   std::cout << "From the following " << point_set.size() << " points "
             << std::endl;
   std::copy(point_set.begin(), point_set.end(), out);
   std::cout << std::endl;

   CGAL::random_polygon_2(point_set.size(), std::back_inserter(polygon),
                          point_set.begin());
   std::cout << "The following simple polygon was made: " << std::endl;
```

```
    std::cout << polygon << std::endl;
    return 0;
}
```

File: examples/Generator/random_polygon.cpp

CGAL::random_selection

Function

Definition

random_selection chooses n items at random from a random access iterator range which is useful to produce degenerate input data sets with multiple entries of identical items.

#include <CGAL/random_selection.h>

template <class RandomAccessIterator, class Size, class OutputIterator, class Random>
OutputIterator random_selection(RandomAccessIterator first, RandomAccessIterator last, Size n, OutputIterator result, Random& rnd = default_random)

chooses a random item from the range $[first, last)$ and writes it to *result*, each item from the range with equal probability, and repeats this n times, thus writing n items to *result*. A single random number is needed from *rnd* for each item. Returns the value of *result* after inserting the n items.
Precondition: *Random* is a random number generator type as provided by the STL or by *Random*.

See Also

CGAL::perturb_points_2 . . . . page 4359

Class

CGAL::Random

Definition

The class *Random* is a random numbers generator. It generates uniformly distributed random *bool*s, *int*s and *double*s. It can be used as the random number generating function object in the STL algorithm *random_shuffle*.

Instances of can be seen as input streams. Different streams are *independent* of each other, i.e. the sequence of numbers from one stream does *not* depend upon how many numbers were extracted from the other streams. At each time, an instance has a *state* that uniquely determines the subsequent numbers being produced.

It can be very useful, e.g. for debugging, to reproduce a sequence of random numbers. This can be done by either initialising with a fixed seed, or by using the state functions as described below.

#include <CGAL/Random.h>

Types

Random:: State	The State type.

Creation

Random random;	introduces a variable *random* of type *Random*. The seed is chosen “randomly”, depending on the system time.
Random random(unsigned int seed);	introduces a variable *random* of type *Random* and initializes its internal state using *seed*. Equal values for *seed* result in equal sequences of random numbers.

Operations

bool	*random.get_bool()*	returns a random *bool*.
template <int b> *int*	*random.get_bits()*	returns a random *int* value from the interval $[0,2\hat{}b)$. This is supposed to be efficient.
int	*random.get_int(int lower, int upper)*	returns a random *int* from the interval $[lower,upper)$.
double	*random.get_double(double lower = 0.0, double upper = 1.0)*	returns a random *double* from the interval $[lower,upper)$.

Distributions

The following member functions are a 1-to-1 correspondence to some distributions from the boost random library.

template <typename IntType>
IntType *random.uniform_smallint(IntType lower=0, IntType upper=9)*

returns a random *IntType* from the interval [*lower*,*upper*]. *IntType* can be an integral type as *int*, *std::ptrdiff_t*, *std::size_t*,etc. **Warning: In contrast to** *get_int* **this function may return** *upper***.**

template <typename IntType>
IntType *random.uniform_int(IntType lower=0, IntType upper=9)*

returns a random *IntType* from the interval [*lower*,*upper*]. *IntType* can be an integral type as *int*, *std::ptrdiff_t*, *std::size_t*,etc. **Warning: In contrast to** *get_int* **this function may return** *upper***.**

template <typename RealType>
Realtype *random.uniform_real(RealType lower = 0.0, RealType upper = 1.0)*

returns a random *RealType* from the interval [*lower*,*upper*). *RealType* can be *float*, *double*, etc.

template <typename RealType>
RealType *random.uniform_01()* returns a random *RealType* from the interval $[0,1)$. *RealType* can be *float*, *double*, etc.

template <typename IntType>
IntType *random(IntType upper)*

returns *randomuniform_int<IntType>(0, upper-1)*.

Seed and State Functions

unsigned int *random.get_seed() const*

returns the seed used for initialization.

void *random.save_state(State& state) const*

saves the current internal state in *state*.

void *random.restore_state(State state)*

restores the internal state from *state*.

Equality Test

bool	*random* == *random2*	returns *true*, iff *random* and *random2* have equal internal states.

Implementation

We use the boost random library function *boost::rand48* to generate the random numbers.

See Also

CGAL::default_random . . . . page 4358

advanced

RandomConvexSetTraits_2

Definition

The concept RandomConvexSetTraits_2describes the requirements of the traits class for the function *random_convex_set_2*.

Has Models

Types

RandomConvexSetTraits_2:: Point_2	point class.
RandomConvexSetTraits_2:: FT	class used for doing computations on point and vector coordinates (has to fulfill field type requirements).
RandomConvexSetTraits_2:: Sum	AdaptableBinaryFunction class: *Point_2* $\times$ *Point_2* $\rightarrow$ *Point_2*. It returns the point that results from adding the vectors corresponding to both arguments.
RandomConvexSetTraits_2:: Scale	AdaptableBinaryFunction class: *Point_2* $\times$ *FT* $\rightarrow$ *Point_2*. *Scale(p,k)* returns the point that results from scaling the vector corresponding to p by a factor of k.
RandomConvexSetTraits_2:: Max_coordinate	AdaptableUnaryFunction class: *Point_2* $\rightarrow$ *FT*. *Max_coordinate(p)* returns the coordinate of p with largest absolute value.
RandomConvexSetTraits_2:: Angle_less	AdaptableBinaryFunction class: *Point_2* $\times$ *Point_2* $\rightarrow$ *bool*. It returns *true*, iff the angle of the direction corresponding to the first argument with respect to the positive x-axis is less than the angle of the direction corresponding to the second argument.

Operations

Point_2	*t.origin() const*	return origin (neutral element for the *Sum* operation).

advanced

Concept

RandomPolygonTraits_2

Definition

The concept RandomPolygonTraits_2 describes the requirements for the traits class used by the function *random_polygon_2*.

Has Models

The CGAL kernels.

Types

RandomPolygonTraits_2:: FT	The coordinate type of the points of the polygon (*i.e.*, a field type)
RandomPolygonTraits_2:: Point_2	The point type of the polygon.
RandomPolygonTraits_2:: Orientation_2	Predicate object type that determines the orientation of three points. It must provide *Orientation operator()(Point_2 p, Point_2 q, Point_2 r)* that returns *LEFT_TURN*, if r lies to the left of the oriented line l defined by p and q, returns *RIGHT_TURN* if r lies to the right of l, and returns *COLLINEAR* if r lies on l.
RandomPolygonTraits_2:: Less_xy_2	Binary predicate object type comparing *Point_2*s lexicographically. It must provide *bool operator()(Point_2 p, Point_2 q)* that returns *true* iff $p <_{xy} q$. We have $p <_{xy} q$, iff $p_x < q_x$ or $p_x = q_x$ and $p_y < q_y$, where p_x and p_y denote the x and y coordinates of point p, resp.

Operations

The following two member functions returning instances of the above predicate object types are required.

Less_xy_2	*t.less_xy_2_object()*
Orienation_2	*t.orientation_2_object()*

advanced

CGAL::Random_convex_set_traits_2<Kernel>

Definition

The class *Random_convex_set_traits_2<Kernel>*serves as a traits class for the function *random_convex_set_2*.

#include <CGAL/Random_convex_set_traits_2.h>

Is Model for the Concepts

RandomConvexSetTraits_2 . . . . page 4377

Types

typedef Kernel::Point_2

	Point_2;	
typedef Kernel::FT	*FT;*	
Random_convex_set_traits_2<Kernel>:: Sum		function object class derived from *std::binary_function< Point_2, Point_2, Point_2>*
Random_convex_set_traits_2<Kernel>:: Scale		function object class derived from *std::binary_function< Point_2, Point_2, Point_2>*

Random_convex_set_traits_2<Kernel>:: Max_coordinate

function object class derived from *std::unary_function< Point_2, FT>*

Random_convex_set_traits_2<Kernel>:: Angle_less

function object class derived from *std::binary_function< Point_2, Point_2, bool>*

Creation

Random_convex_set_traits_2<Kernel> t;	default constructor

Operations

Point_2	*t.origin() const*	returns CGAL::ORIGIN.

advanced

Class

CGAL::Random_points_in_ball_d<Point_d>

Definition

The class *Random_points_in_ball_d<Point_d>* is an input iterator creating points uniformly distributed in an open ball in any dimension.

Is Model for the Concepts

InputIterator
PointGenerator . . . . page 4366

#include <CGAL/point_generators_d.h>

Types

typedef std::input_iterator_tag	*iterator_category;*
typedef Point_d	*value_type;*
typedef std::ptrdiff_t	*difference_type;*
*typedef const Point_d**	*pointer;*
typedef Point_d	*reference;*

Operations

Random_points_in_ball_d<Point_d> *g(int dim, double r);*

g is an input iterator creating points of type *Point_d* uniformly distributed in the open ball in dimension *dim* with radius r, i.e. $|*g| < r$. $2 \cdot dim + 1$ random numbers are needed from *rnd* for each point.

See Also

CGAL::cpp0x::copy_n . . . . page 4174
CGAL::Counting_iterator . . . . page **??**
CGAL::Random_points_in_disc_2<Point_2, Creator> . . . . page 4383
CGAL::Random_points_in_sphere_3<Point_3, Creator> . . . . page 4384
CGAL::Random_points_in_cube_d<Point_d> . . . . page 4382
CGAL::Random_points_on_sphere_d<Point_d> . . . . page 4389

CGAL::Random_points_in_cube_3<Point_3, Creator>

Class

Definition

The class *Random_points_in_cube_3<Point_3, Creator>* is an input iterator creating points uniformly distributed in a half-open cube. The default *Creator* is *Creator_uniform_3<Kernel_traits<P>::Kernel::RT,P>*.

Is Model for the Concepts

InputIterator
PointGenerator . . . . page 4366

#include <CGAL/point_generators_3.h>

Types

typedef std::input_iterator_tag	*iterator_category;*
typedef Point_3	*value_type;*
typedef std::ptrdiff_t	*difference_type;*
*typedef const Point_3**	*pointer;*
typedef Point_3	*reference;*

Operations

Random_points_in_cube_3<Point_3, Creator> g(double a, Random& rnd = default_random);

g is an input iterator creating points of type *Point_3* uniformly distributed in the half-open cube with side length $2a$, centered at the origin, i.e. $\forall p = *g : -a \leq p.x(), p.y(), p.z() < a$. Three random numbers are needed from *rnd* for each point.

See Also

CGAL::cpp0x::copy_n . . . . page 4174
CGAL::Counting_iterator . . . . page **??**
CGAL::Random_points_in_square_2<Point_2, Creator> . . . . page 4385
CGAL::Random_points_in_sphere_3<Point_3, Creator> . . . . page 4384
CGAL::Random_points_on_sphere_3<Point_3, Creator> . . . . page 4388
std::random_shuffle

Class

CGAL::Random_points_in_cube_d<Point_d>

Definition

The class *Random_points_in_cube_d<Point_d>* is an input iterator creating points uniformly distributed in an half-open cube.

Is Model for the Concepts

InputIterator
PointGenerator . . . . page 4366

#include <CGAL/point_generators_d.h>

Types

typedef std::input_iterator_tag	*iterator_category;*
typedef Point_d	*value_type;*
typedef std::ptrdiff_t	*difference_type;*
*typedef const Point_d**	*pointer;*
typedef Point_d	*reference;*

Operations

Random_points_in_cube_d<Point_d> g(int dim, double a, Random& rnd = default_random);

g is an input iterator creating points of type *Point_d* uniformly distributed in the half-open cube of dimension *dim* with side length $2a$, centered at the origin. For every point $p = *g$ and for all $i < dim$ we have $-a \leq p[i] < a$. *dim* random numbers are needed from *rnd* for each point.

See Also

CGAL::cpp0x::copy_n . . . . page 4174
CGAL::Counting_iterator . . . . page ??
CGAL::Random_points_in_square_2<Point_2, Creator> . . . . page 4385
CGAL::Random_points_in_cube_3<Point_3, Creator> . . . . page 4381
CGAL::Random_points_in_ball_d<Point_d> . . . . page 4380
CGAL::Random_points_on_sphere_d<Point_d> . . . . page 4389

CGAL::Random_points_in_disc_2<Point_2, Creator>

Class

Definition

The class *Random_points_in_disc_2<Point_2, Creator>* is an input iterator creating points uniformly distributed in an open disc. The default *Creator* is *Creator_uniform_2<Kernel_traits<P>::Kernel::RT,P>*.

Is Model for the Concepts

InputIterator
PointGenerator . . . . page 4366

#include <CGAL/point_generators_2.h>

Types

typedef std::input_iterator_tag	*iterator_category;*
typedef Point_2	*value_type;*
typedef std::ptrdiff_t	*difference_type;*
*typedef const Point_2**	*pointer;*
typedef Point_2	*reference;*

Operations

Random_points_in_disc_2<Point_2, Creator> g(double r, Random& rnd = default_random);

g is an input iterator creating points of type *Point_2* uniformly distributed in the open disc with radius *r*, i.e. $|*g| < r$. Two random numbers are needed from *rnd* for each point.

See Also

CGAL::cpp0x::copy_n . . . . page 4174
CGAL::Counting_iterator . . . . page **??**
CGAL::Points_on_segment_2<Point_2> . . . . page 4361
CGAL::Random_points_in_square_2<Point_2, Creator> . . . . page 4385
CGAL::Random_points_on_circle_2<Point_2, Creator> . . . . page 4386
CGAL::Random_points_on_segment_2<Point_2, Creator> . . . . page 4387
CGAL::Random_points_on_square_2<Point_2, Creator> . . . . page 4390
CGAL::Random_points_in_sphere_3<Point_3, Creator> . . . . page 4384 *std::random_shuffle*

Class

CGAL::Random_points_in_sphere_3<Point_3, Creator>

Definition

The class *Random_points_in_sphere_3<Point_3, Creator>* is an input iterator creating points uniformly distributed in an open sphere. The default *Creator* is *Creator_uniform_3<Kernel_traits<P>::Kernel::RT,P>*.

Is Model for the Concepts

InputIterator
PointGenerator . . . . page 4366

#include <CGAL/point_generators_3.h>

Types

typedef std::input_iterator_tag	*iterator_category;*
typedef Point_3	*value_type;*
typedef std::ptrdiff_t	*difference_type;*
*typedef const Point_3**	*pointer;*
typedef Point_3	*reference;*

Operations

Random_points_in_sphere_3<Point_3, Creator> g(double r, Random& rnd = default_random);

g is an input iterator creating points of type *Point_3* uniformly distributed in the open sphere with radius *r*, i.e. $|*g| < r$. Three random numbers are needed from *rnd* for each point.

See Also

CGAL::cppox::copy_n . . . . page 4174
CGAL::Counting_iterator . . . . page ??
CGAL::Random_points_in_disc_2<Point_2, Creator> . . . . page 4383
CGAL::Random_points_in_cube_3<Point_3, Creator> . . . . page 4381
CGAL::Random_points_on_sphere_3<Point_3, Creator> . . . . page 4388
std::random_shuffle

CGAL::Random_points_in_square_2<Point_2, Creator>

Definition

The class *Random_points_in_square_2<Point_2, Creator>* is an input iterator creating points uniformly distributed in a half-open square. The default *Creator* is *Creator_uniform_2<Kernel_traits<P>::Kernel::RT,P>*.

Is Model for the Concepts

InputIterator
PointGenerator . . . . page 4366

#include <CGAL/point_generators_2.h>

Types

typedef std::input_iterator_tag	*iterator_category;*
typedef Point_2	*value_type;*
typedef std::ptrdiff_t	*difference_type;*
*typedef const Point_2**	*pointer;*
typedef Point_2	*reference;*

Operations

Random_points_in_square_2<Point_2, Creator> g(double a, Random& rnd = default_random);

g is an input iterator creating points of type *Point_2* uniformly distributed in the half-open square with side length $2a$, centered at the origin, i.e. $\forall p = *g : -a \leq p.x() < a$ and $-a \leq p.y() < a$. Two random numbers are needed from *rnd* for each point.

See Also

CGAL::cpp0x::copy_n . . . . page 4174
CGAL::Counting_iterator . . . . page **??**
CGAL::Points_on_segment_2<Point_2> . . . . page 4361
CGAL::Random_points_in_disc_2<Point_2, Creator> . . . . page 4383
CGAL::Random_points_on_segment_2<Point_2, Creator> . . . . page 4387
CGAL::Random_points_on_square_2<Point_2, Creator> . . . . page 4390
CGAL::Random_points_in_cube_3<Point_3, Creator> . . . . page 4381
std::random_shuffle

Class

CGAL::Random_points_on_circle_2<Point_2, Creator>

Definition

The class *Random_points_on_circle_2<Point_2, Creator>* is an input iterator creating points uniformly distributed on a circle. The default *Creator* is *Creator_uniform_2<Kernel_traits<P>::Kernel::RT,P>*. The generated points are computed using floating point arithmetic, whatever the Kernel is, thus they are on the circle/-sphere only up to rounding errors.

Is Model for the Concepts

InputIterator
PointGenerator . . . . page 4366

#include <CGAL/point_generators_2.h>

Types

typedef std::input_iterator_tag	*iterator_category;*
typedef Point_2	*value_type;*
typedef std::ptrdiff_t	*difference_type;*
*typedef const Point_2**	*pointer;*
typedef Point_2	*reference;*

Operations

Random_points_on_circle_2<Point_2, Creator> g(double r, Random& rnd = default_random);

g is an input iterator creating points of type *Point_2* uniformly distributed on the circle with radius r, i.e. $|*g| == r$. A single random number is needed from *rnd* for each point.

See Also

See Also

CGAL::cpp0x::copy_n . . . . page 4174
CGAL::Counting_iterator . . . . page **??**
CGAL::Points_on_segment_2<Point_2> . . . . page 4361
CGAL::Random_points_in_disc_2<Point_2, Creator> . . . . page 4383
CGAL::Random_points_in_square_2<Point_2, Creator> . . . . page 4385
CGAL::Random_points_on_segment_2<Point_2, Creator> . . . . page 4387
CGAL::Random_points_on_square_2<Point_2, Creator> . . . . page 4390
CGAL::Random_points_on_sphere_3<Point_3, Creator> . . . . page 4388
std::random_shuffle

CGAL::Random_points_on_segment_2<Point_2, Creator>

Class

Definition

The class *Random_points_on_segment_2<Point_2, Creator>* is an input iterator creating points uniformly distributed on a segment. The default *Creator* is *Creator_uniform_2<Kernel_traits<P>::Kernel::RT,P>*.

Is Model for the Concepts

InputIterator
PointGenerator . . . page 4366

#include <CGAL/point_generators_2.h>

Types

typedef std::input_iterator_tag	*iterator_category;*
typedef Point_2	*value_type;*
typedef std::ptrdiff_t	*difference_type;*
*typedef const Point_2**	*pointer;*
typedef Point_2	*reference;*

Operations

Random_points_on_segment_2<Point_2, Creator> g(Point_2 p, Point_2 q, Random& rnd = default_random);

g is an input iterator creating points of type *Point_2* uniformly distributed on the segment from p to q (excluding q), i.e. $*g == (1-\lambda)\,p+\lambda q$ where $0 \leq \lambda < 1$. A single random number is needed from *rnd* for each point.
Requirement: The expressions *to_double(p.x())* and *to_double(p.y())* must result in the respective *double* representation of the coordinates of p, and similarly for q.

See Also

CGAL::cpp0x::copy_n . . . page 4174
CGAL::Counting_iterator . . . page ??
CGAL::Points_on_segment_2<Point_2> . . . page 4361
CGAL::Random_points_in_disc_2<Point_2, Creator> . . . page 4383
CGAL::Random_points_in_square_2<Point_2, Creator> . . . page 4385
CGAL::Random_points_on_circle_2<Point_2, Creator> . . . page 4386
CGAL::Random_points_on_square_2<Point_2, Creator> . . . page 4390
std::random_shuffle

CGAL::Random_points_on_sphere_3<Point_3, Creator>

Definition

The class *Random_points_on_sphere_3<Point_3, Creator>* is an input iterator creating points uniformly distributed on a sphere. The default *Creator* is *Creator_uniform_3<Kernel_traits<P>::Kernel::RT,P>*. The generated points are computed using floating point arithmetic, whatever the Kernel is, thus they are on the circle/-sphere only up to rounding errors.

Is Model for the Concepts

InputIterator
PointGenerator .... page 4366

#include <CGAL/point_generators_3.h>

Types

typedef std::input_iterator_tag	*iterator_category;*
typedef Point_3	*value_type;*
typedef std::ptrdiff_t	*difference_type;*
*typedef const Point_3**	*pointer;*
typedef Point_3	*reference;*

Operations

Random_points_on_sphere_3<Point_3, Creator> g(double r, Random& rnd = default_random);

g is an input iterator creating points of type *Point_3* uniformly distributed on the boundary of a sphere with radius r, i.e. $|*g| == r$. Two random numbers are needed from *rnd* for each point.

See Also

CGAL::cpp0x::copy_n .... page 4174
CGAL::Counting_iterator .... page **??**
CGAL::Random_points_on_circle_2<Point_2, Creator> .... page 4386
CGAL::Random_points_in_cube_3<Point_3, Creator> .... page 4381
CGAL::Random_points_in_sphere_3<Point_3, Creator> .... page 4384
std::random_shuffle

CGAL::Random_points_on_sphere_d<Point_d>

Definition

The class *Random_points_on_sphere_d<Point_d>* is an input iterator creating points uniformly distributed on a sphere.

The generated points are computed using floating point arithmetic, whatever the Kernel is, thus they are on the sphere only up to rounding errors.

Is Model for the Concepts

InputIterator
PointGenerator . . . . page 4366

#include <CGAL/point_generators_d.h>

Types

typedef std::input_iterator_tag	*iterator_category;*
typedef Point_d	*value_type;*
typedef std::ptrdiff_t	*difference_type;*
*typedef const Point_d**	*pointer;*
typedef Point_d	*reference;*

Operations

Random_points_on_sphere_d<Point_d> g(int dim, double r, Random& rnd = default_random);

g is an input iterator creating points of type *Point_d* uniformly distributed on a sphere in dimension *dim* with radius r, i.e. $|*g| == r$. $2 \cdot dim$ random numbers are needed from *rnd* for each point.

See Also

CGAL::cpp0x::copy_n . . . . page 4174
CGAL::Counting_iterator . . . . page **??**
CGAL::Random_points_on_circle_2<Point_2, Creator> . . . . page 4386
CGAL::Random_points_on_sphere_3<Point_3, Creator> . . . . page 4388
CGAL::Random_points_in_cube_d<Point_d> . . . . page 4382
CGAL::Random_points_in_ball_d<Point_d> . . . . page 4380

Class

CGAL::Random_points_on_square_2<Point_2, Creator>

Definition

The class *Random_points_on_square_2<Point_2, Creator>* is an input iterator creating points uniformly distributed in the boundary of a square. The default *Creator* is *Creator_uniform_2<Kernel_traits<P> ::Kernel::RT,P>*.

Is Model for the Concepts

InputIterator
PointGenerator . . . . page 4366

#include <CGAL/point_generators_2.h>

Types

typedef std::input_iterator_tag	*iterator_category;*
typedef Point_2	*value_type;*
typedef std::ptrdiff_t	*difference_type;*
*typedef const Point_2**	*pointer;*
typedef Point_2	*reference;*

Operations

Random_points_on_square_2<Point_2, Creator> g(double a, Random& rnd = default_random);

g is an input iterator creating points of type *Point_2* uniformly distributed on the boundary of the square with side length $2a$, centered at the origin, i.e. $\forall p = *g$: one coordinate is either a or $-a$ and for the other coordinate c holds $-a \le c < a$. A single random number is needed from *rnd* for each point.

See Also

CGAL::cpp0x::copy_n . . . . page 4174
CGAL::Counting_iterator . . . . page **??**
CGAL::Points_on_segment_2<Point_2> . . . . page 4361
CGAL::Random_points_in_disc_2<Point_2, Creator> . . . . page 4383
CGAL::Random_points_in_square_2<Point_2, Creator> . . . . page 4385
CGAL::Random_points_on_circle_2<Point_2, Creator> . . . . page 4386
CGAL::Random_points_on_segment_2<Point_2, Creator> . . . . page 4387
std::random_shuffle

Chapter 78

Profiling tools, Hash Map, Union-find, Modifiers

Lutz Kettner, Sylvain Pion, and Michael Seel

Contents

78.1 Timers

CGAL provides classes for measuring the user process time and the real time. The class *CGAL::Timer* is the version for the user process time and the class *CGAL::Real_timer* is the version for the real time.

Instantiations of both classes are objects with a state. The state is either *running* or it is *stopped*. The state of an object *t* is controlled with *t.start()* and *t.stop()* . The timer counts the time elapsed since its creation or last reset. It counts only the time where it is in the running state. The time information is given in seconds. The timer counts also the number of intervals it was running, i.e. it counts the number of calls of the *start()* member function since the last reset. If the reset occurs while the timer is running it counts as the first interval.

78.2 Memory Size

CGAL provides access to the memory size used by the program with the *CGAL::Memory_sizer* class. Both the virtual memory size and the resident size are available (the resident size does not account for swapped out memory nor for the memory which is not yet paged-in).

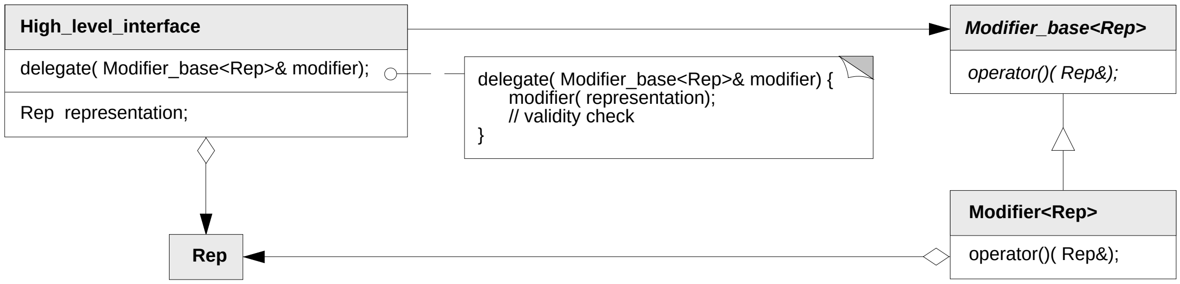


Figure 78.1: Class diagram for the modifier. It illustrates the safe access to an internal representation through an high-level interface.

78.3 Profiling

CGAL provides a way to count the number of times a given line of code is executed during the execution of a program. Such *CGAL::Profile_counter* counters can be added at critical place in the code, and at the end of the execution of a program, the count is printed on *std::cerr*. A macro *CGAL_PROFILER* can be used to conveniently place these counters anywhere. They are disabled by default and activated by the global macro *CGAL_PROFILE*.

78.4 Unique Hash Map

The class *Unique_hash_map* implements an injective mapping between a set of unique keys and a set of data values. This is implemented using a chained hashing scheme and access operations take $O(1)$ expected time. Such a mapping is useful, for example, when keys are pointers, handles, iterators or circulators that refer to unique memory locations. In this case, the default hash function is *Handle_hash_function*.

78.5 Union-find

CGAL also provides a class *Union_find* that implements a partition of values into disjoint sets. This is implemented with union by rank and path compression. The running time for m set operations on n elements is $O(n\alpha(m,n))$ where $\alpha(m,n)$ is the extremely slowly growing inverse of Ackermann's function.

78.6 Protected Access to Internal Representations

High level data structures typically maintain integrity of an internal data representation, which they protect from the user. A minimal while complete interface of the data structure allows manipulations in the domain of valid representations. Additional operations might benefit from being allowed to access the internal data representation directly. An example are intermediate steps within an algorithm where the internal representation would be invalid. We present a general method to accomplish access in a safe manner, such that the high level data structures can guarantee validity after the possibly compromising algorithm has finished its work. An example are polyhedral surfaces in the Basic Library, where a construction process like for a file scanner could be performed more efficiently on the internal halfedge data structure than by using the high-level Euler operators of the polyhedron.

The solution provided here is inspired by the strategy pattern [GHJV95], though it serves a different intent, see Figure 78.1. The abstract base class *Modifier_base<R>* declares a pure virtual member function *operator()* that accepts a single reference parameter of the internal representation type. The member function *delegate()* of the high-level interface calls this *operator()* with its internal representation. An actual modifier implements this virtual function, thus gaining access to the internal representation. Once, the modifier has finished its work, the member function *delegate()* is back in control and can check the validity of the internal representation. Summarizing, a user can implement and apply arbitrary functions based on the internal representation and keeps the benefit if a protected high-level interface. User provided modifiers must in any case return a valid internal representation or the checker in the high-level interface is allowed (and supposed) to abort the program. The indirection via the virtual function invocation is negligible for operations that consists of more than a pointer update or integer addition.

Profiling tools, Hash Map, Union-find, Modifiers Reference Manual

Lutz Kettner, Sylvain Pion, and Michael Seel

This chapter decribes classes for measuring user process time and real time as well as the memory size. It also provides a profile counter tool.

A hash map *CGAL::Unique_hash_map* is offered that is specialized on unique hash values of type *std::size_t*, i.e., it is particularly useful for pointers, handles, iterators, and circulators as key values.

Furthermore, a union-find data structure and the modifier base class is documented.

78.7 Classified Reference Pages

Concepts

Classes

78.8 Alphabetical List of Reference Pages

CGAL::Handle_hash_function

Class

Definition

The class *Handle_hash_function* is a model for the *UniqueHasFunction* concept. It is applicable for all key types with pointer-like functionality, such as handles, iterators, and circulators. Specifically, for a *key* value the expression `&`**key* must return a unique address.

#include <CGAL/Handle_hash_function.h>

Is Model for the Concepts

UniqueHashFunction . . . . page 4407

Creation

Handle_hash_function hash;	default constructor.

Operations

template <class Handle>		
std::size_t	*hash(Handle key)*	returns unique hash value for any *Handle* type for which `&`**key* gives a unique address. *Requirement*: The type *std::iterator_traits<Handle>::value_type* has to be defined (which it is already for pointers, handles, iterators, and circulators).

See Also

CGAL::Unique_hash_map<Key,Data,UniqueHashFunction> . . . . page 4405

Implementation

Plain type cast of `&`**key* to *std::size_t* and devided by the size of the *std::iterator_traits<Handle>::value_type* to avoid correlations with the internal table size, which is a power of two.

Class

CGAL::Memory_sizer

Definition

The class *Memory_sizer* allows to measure the memory size used by the process. Both the virtual memory size and the resident size are available (the resident size does not account for swapped out memory nor for the memory which is not yet paged-in).

#include <CGAL/Memory_sizer.h>

Types

typedef std::size_t *size_type;*

Creation

Memory_sizer m; Default constructor.

Operations

size_type

m.virtual_size() const

Returns the virtual memory size in bytes.

size_type

m.resident_size() const

Returns the resident memory size in bytes.

Implementation

Accessing this information requires the use of non-portable code. Currently, there is support for Linux platforms, the Microsoft and Intel compiler on Windows, as well as Mac OS X. If a platform is not supported, the two member functions return 0.

CGAL::Modifier_base<R>

Class

Definition

Modifier_base<R> is an abstract base class providing the interface for any modifier. A modifier is a function object derived from *Modifier_base<R>* that implements the pure virtual member function *operator()*, which accepts a single reference parameter *R*`&` on which the modifier is allowed to work. *R* is the type of the internal representation that is to be modified.

#include <CGAL/Modifier_base.h>

Types

typedef R	*Representation;*	the internal representation type.

Operations

virtual void	*modifier.operator()(R*`&` *rep)*	*Postcondition*: *rep* is a valid representation.

Example

The following fragment defines a class `A` with an internal representation `i` of type `int`. It provides a member function `delegate()`, which gives a modifier access to the internal variable and checks validity thereafter. The example modifier sets the internal variable to 42. The example function applies the modifier to an instance of class `A`.

```
class A {
    int i;  // protected internal representation
public:
    void delegate( CGAL::Modifier_base<int>& modifier) {
        modifier(i);
        CGAL_postcondition( i > 0);  // check validity
    }
};

struct Modifier : public CGAL::Modifier_base<int> {
    void operator()( int& rep) { rep = 42;}
};

void use_it() {
    A a;
    Modifier m;
    a.delegate(m);  // a.i == 42 and A has checked that A::i > 0.
}
```

Class

CGAL::Profile_counter

Definition

#include <CGAL/Profile_counter.h>

The class *Profile_counter* provides a way to count the number of times a given line of code is executed during the execution of a program, and print this number at the end of the execution of the program. Such counters can be added at critical places in the code, and at the end of the execution of a program, the count is printed on *std::cerr*, together with an identification string passed to the constructor. The macro *CGAL_PROFILER* can be used to conveniently place these counters anywhere. They are disabled by default and activated by the global macro *CGAL_PROFILE*.

Creation

Profile_counter pc(std::string s); The internal counter is initialized to 0, and the string *s* is stored for further printing by the destructor.

Profile_counter pc; The value of the counter is printed to *std::cerr* together with the string.

Operations

void $++pc$ Increments the internal counter.

#define CGAL_PROFILER(MSG) If *CGAL_PROFILE* is not defined, then *CGAL_PROFILER* is defined to an empty statement. Otherwise, it is defined to *static CGAL::Profile_counter tmp(MSG); ++tmp;*.

```
#define CGAL_PROFILE

#include <CGAL/Profile_counter.h>

int main()
{
  for (int i=0; i<10; ++i)
  {
    CGAL_PROFILER("iterations of the for-loop");
  }
  return 0;
}
```

File: `examples/Profiling_tools/Profile_counter.cpp`

will print at exit:

```
[CGAL::Profile_counter]     10 iterations of the for-loop
```

CGAL::Real_timer

Definition

#include <CGAL/Real_timer.h>

The class *Real_timer* is a timer class for measuring real time. A timer *t* of type *Real_timer* is an object with a state. It is either *running* or it is *stopped*. The state is controlled with *t.start()* and *t.stop()*. The timer counts the time elapsed since its creation or last reset. It counts only the time where it is in the running state. The time information is given in seconds. The timer counts also the number of intervals it was running, i.e. it counts the number of calls of the *start()* member function since the last reset. If the reset occures while the timer is running it counts as the first interval.

Creation

Real_timer t;	state is *stopped.*

Operations

void	*t.start()*	*Precondition*: state is *stopped*.
void	*t.stop()*	*Precondition*: state is *running*.
void	*t.reset()*	reset timer to zero. The state is unaffected.
bool	*t.is_running()*	*true* if the current state is running.
double	*t.time()*	real time in seconds, or 0 if the underlying system call failed.
int	*t.intervals()*	number of start/stop-intervals since the last reset.
double	*t.precision()*	smallest possible time step in seconds, or -1 if the system call failed.
double	*t.max()*	maximal representable time in seconds.

Implementation

The timer class is based in the C function *gettimeofday()* on POSIX systems and the C function *_ftime()* on MS Visual C++. The system calls to these timers might fail, in which case a warning message will be issued through the CGAL error handler and the functions return with the error codes indicated above. The *precision* method computes the precision dynamically at runtime at its first invocation.

CGAL::Timer

Definition

The class *Timer* is a timer class for measuring user process time. A timer *t* of type *Timer* is an object with a state. It is either *running* or it is *stopped*. The state is controlled with *t.start()* and *t.stop()*. The timer counts the time elapsed since its creation or last reset. It counts only the time where it is in the running state. The time information is given in seconds. The timer counts also the number of intervals it was running, i.e. it counts the number of calls of the *start()* member function since the last reset. If the reset occures while the timer is running it counts as the first interval.

#include <CGAL/Timer.h>

Creation

Timer t;	state is *stopped.*

Operations

void	*t.start()*	*Precondition*: state is *stopped*.
void	*t.stop()*	*Precondition*: state is *running*.
void	*t.reset()*	reset timer to zero. The state is unaffected.
bool	*t.is_running()*	*true* if the current state is running.
double	*t.time()*	user process time in seconds, or 0 if the underlying system call failed.
int	*t.intervals()*	number of start/stop-intervals since the last reset.
double	*t.precision()*	smallest possible time step in seconds, or -1 if the system call failed.
double	*t.max()*	maximal representable time in seconds.

Implementation

The timer class is based in the C function *std::clock()* on PC systems and the C function *getrusage()* on standard POSIX systems. The counter for the *std::clock()* based solution might wrap around (overflow) after only about 36 minutes. This won't happen on POSIX systems. The system calls to these timers might fail, in which case a warning message will be issued through the CGAL error handler and the functions return with the error codes indicated above. The *precision* method computes the precision dynamically at runtime at its first invocation.

CGAL::Union_find<T,A>

Class

Definition

An instance *P* of the data type *Union_find<T,A>* is a partition of values of type *T* into disjoint sets. The template parameter *A* has to be a model of the allocator concept as defined in the C++ standard. It has a default argument *CGAL_ALLOCATOR(T)*.

#include <CGAL/Union_find.h>

Types

Union_find<T,A>:: value_type	values stored in items (equal to *T*).
Union_find<T,A>:: handle	handle to values.
Union_find<T,A>:: iterator	iterator over values.

There are also constant versions *const_handle* and *const_iterator*.

Union_find<T,A>:: allocator	allocator.

Creation

Union_find<T,A> P;	creates an instance *P* of type *Union_find<T,A>* and initializes it to the empty partition.

Operations

allocator	*P.get_allocator()*	the allocator of *P*.
std::size_t	*P.number_of_sets()*	returns the number of disjoint sets of *P*.
std::size_t	*P.size()*	returns the number of values of *P*.
std::size_t	*P.bytes()*	returns the memory consumed by *P*.
std::size_t	*P.size(const_handle p)*	returns the size of the set containing *p*.
void	*P.clear()*	reinitializes *P* to an empty partition.
handle	*P.make_set(T x)*	creates a new singleton set containing *x* and returns a handle to it.
handle	*P.push_back(T x)*	same as *make_set(x)*.

template <class Forward_iterator>

void	*P.insert(Forward_iterator first, Forward_iterator beyond)*	
		insert the range of values referenced by *[first,beyond)*. *Requirement*: value type of *Forward_iterator* is T.
handle	*P.find(handle p)*	
const_handle	*P.find(const_handle p)*	returns a canonical handle of the set that contains p, i.e., *P.same_set(p,q)* iff *P.find(p)* and *P.find(q)* return the same handle. *Precondition*: p is a handle in P.
void	*P.unify_sets(handle p, handle q)*	
		unites the sets of partition P containing p and q. *Precondition*: p and q are in P.
bool	*P.same_set(const_handle p, const_handle q)*	
		returns true iff p and q belong to the same set of P. *Precondition*: p and q are in P.
iterator	*P.begin()*	returns an iterator pointing to the first value of P.
iterator	*P.end()*	returns an iterator pointing beyond the last value of P.

Implementation

Union_find<T,A> is implemented with union by rank and path compression. The running time for m set operations on n elements is $O(n\alpha(m,n))$ where $\alpha(m,n)$ is the extremely slow growing inverse of Ackermann's function.

CGAL::Unique_hash_map<Key,Data,UniqueHashFunction>

Definition

An instance *map* of the parameterized data type *Unique_hash_map<Key,Data,UniqueHashFunction>* is an injective mapping from the set of keys of type *Key* to the set of variables of type *Data*. New keys can be inserted at any time, however keys cannot be individually deleted.

An object *hash* of the type *UniqueHashFunction* returns a unique integer index *hash(key)* of type *std::size_t* for all objects *key* stored in *map*. The template parameter has as default the *Handle_hash_function* that hashes all types of pointers, handles, iterators, and circulators.

All variables are initialized to *default_data*, a value of type *Data* specified in the definition of *map*.

#include <CGAL/Unique_hash_map.h>

Types

Unique_hash_map<Key,Data,UniqueHashFunction>:: Key	the *Key* type.
Unique_hash_map<Key,Data,UniqueHashFunction>:: Data	the *Data* type.
Unique_hash_map<Key,Data,UniqueHashFunction>:: Hash_function	the unique hash function type.

In compliance with STL, the types *key_type*, *data_type*, and *hasher* are defined as well.

Creation

Unique_hash_map<Key,Data,UniqueHashFunction> map(Data default = Data(),
std::size_t table_size = 1,
Hash_function fct = Hash_function())

creates an injective function *map* from *Key* to the set of unused variables of type *Data*, sets *default_data* to *default*, passes the *table_size* as argument to the internal implementation, and initializes the hash function with *fct*.

Unique_hash_map<Key,Data,UniqueHashFunction> map(Key first1,
Key beyond1,
Data first2,
Data default = Data(),
std::size_t table_size = 1,
Hash_function fct = Hash_function())

creates an injective function *map* from *Key* to the set of unused variables of type *Data*, sets *default_data* to *default*, passes the *table_size* as argument to the internal implementation, initializes the hash function with *fct*, and inserts all keys from the range *[first1,beyond1)*. The data variable for each inserted *key* is initialized with the corresponding value from the range *[first2, first2 + (beyond1-first1))*.
Precondition: The increment operator must be defined for values of type *Key* and for values of type *Data*. *beyond1* must be reachable from *first1* using increments.

Operations

Data	*map.default_value() const*	the current *default_value*.
Hash_function	*map.hash_function() const*	the current hash function.
bool	*map.is_defined(Key key)*	returns true if *key* is defined in *map*. Note that there can be keys defined that have not been inserted explicitly. Their variables are initialized to *default_value*.
void	*map.clear()*	resets *map* to the injective function *map* from *Key* to the set of unused variables of type *Data*. The *default_data* remains unchanged.
void	*map.clear(Data default)*	resets *map* to the injective function *map* from *Key* to the set of unused variables of type *Data* and sets *default_data* to *default*.
Data&	*map.operator[](const Key& key)*	returns a reference to the variable *map(key)*. If *key* has not been inserted into *map* before, *key* is inserted and initialized with *default_value*.
const Data&	*map.operator[](const Key& key) const*	returns a const reference to the variable *map(key)*. If *key* has not been inserted into *map* before, a const reference to the *default_value* is returned. However, *key* is not inserted into *map*.
Data	*map.insert(Key first1, Key beyond1, Data first2)*	inserts all keys from the range *[first1,beyond1)*. The data variable for each inserted *key* is initilized with the corresponding value from the range *[first2, first2 + (beyond1-first1))*. Returns *first2 + (beyond1-first1)*. *Precondition*: The increment operator must be defined for values of type *Key* and for values of type *Data*. *beyond1* must be reachable from *first1* using increments.

See Also

UniqueHashFunction . . . . page 4407
CGAL::Handle_hash_function . . . . page 4397

Implementation

Unique_hash_map is implemented via a chained hashing scheme. Access operations *map[i]* take expected time $O(1)$. The *table_size* parameter passed to chained hashing can be used to avoid unnecessary rehashing when set to the number of expected elements in the map. The design is derived from the STL *hash_map* and the LEDA type *map*. Its specialization on insertion only and unique hash values allow for a more time- and space-efficient implementation, see also [MN00, Chapter 5]. This implementation makes also use of sentinels that lead to defined keys that have not been inserted.

UniqueHashFunction

Concept

Definition

UniqueHashFunction is a concept for a hash function with unique hash values. An instance *hash* for a model of the UniqueHashFunction concept is a function object. It maps objects of its domain type *Key* to the integral image type *std::size_t*. The image values have to be unique for all keys in the domain type *Key*.

Refines

STL concept HashFunction.

Types

typedef std::size_t	*result_type;*	type of the hash value.

Creation

UniqueHashFunction hash(hash2);		copy constructor.
UniqueHashFunction&	*hash = hash2*	assignment.

Operations

std::size_t	*hash(Key key)*	returns unique hash value for the *key* value.

Has Models

CGAL::Handle_hash_function . . . . page 4397

See Also

CGAL::Unique_hash_map<Key,Data,UniqueHashFunction> . . . . page 4405

Chapter 79

IO Streams

Andreas Fabri, Geert-Jan Giezeman, and Lutz Kettner

Contents

All classes in the CGAL kernel provide input and output operators for IO streams. Classes external to CGAL are also supported, by means of *oformat* (Section 79.3). The basic task of such an operator is to produce a representation of an object that can be written as a sequence of characters on devices as a console, a file, or a pipe. In CGAL we distinguish between a raw ascii, a raw binary and a pretty printing format.

enum Mode { *ASCII = 0, BINARY, PRETTY* }*;*

In *ASCII* mode, objects are written as a set of numbers, e.g. the coordinates of a point or the coefficients of a line, in a machine independent format. In *BINARY* mode, data are written in a binary format, e.g. a double is represented as a sequence of four byte. The format depends on the machine. The mode *PRETTY* serves mainly for debugging as the type of the geometric object is written, as well as the data defining the object. For example for a point at the origin with Cartesian double coordinates, the output would be *PointC2(0.0, 0.0)*. At the moment CGAL does not provide input operations for pretty printed data. By default a stream is in ASCII mode.

CGAL provides the following functions to modify the mode of an IO stream.

IO::Mode *set_mode(std::ios& s, IO::Mode m)*

IO::Mode *set_ascii_mode(std::ios& s)*
IO::Mode *set_binary_mode(std::ios& s)*

IO::Mode *set_pretty_mode(std::ios& s)*

The following functions allow to test whether a stream is in a certain mode.

IO::Mode *get_mode(std::ios& s)*

bool *is_ascii(std::ios& s)*
bool *is_binary(std::ios& s)*
bool *is_pretty(std::ios& s)*

79.1 Output Operator

CGAL defines output operators for classes that are derived from the class *ostream*. This allows to write to ostreams as *cout* or *cerr*, as well as to strstreams and fstreams. The output operator is defined for all classes in the CGAL kernel and for the class *Color* as well. Let *os* be an output stream.

ostream& *ostream& os << Class c* Inserts object *c* in the stream *os*. Returns *os*.

Example

```
#include <CGAL/basic.h>
#include <iostream>
#include <fstream>

#include <CGAL/Cartesian.h>
#include <CGAL/Segment_2.h>

typedef CGAL::Point_2< CGAL::Cartesian<double> >     Point;
typedef CGAL::Segment_2< CGAL::Cartesian<double> >   Segment;

int main()
{
    Point p(0,1), q(2,2);
    Segment s(p,q);

    CGAL::set_pretty_mode(std::cout);
    std::cout << p << std::endl << q  << std::endl;

    std::ofstream f("data.txt");
    CGAL::set_binary_mode(f);
    f << s << p ;

    return 1;
}
```

79.2 Input Operator

CGAL defines input operators for classes that are derived from the class *istream*. This allows to read from istreams as *cin*, as well as from strstreams and fstreams. The input operator is defined for all classes in the CGAL kernel. Let *is* be an input stream.

istream&	*istream& is >> Class c*	Extracts object *c* from the stream *is*. Returns *is*.

Example

```
#include <CGAL/basic.h>
#include <iostream>
#include <fstream>

#include <CGAL/Cartesian.h>
#include <CGAL/Segment_2.h>

typedef CGAL::Point_2< CGAL::Cartesian<double> >     Point;
typedef CGAL::Segment_2< CGAL::Cartesian<double> >   Segment;

int
main()
{
    Point p, q;
    Segment s;

    CGAL::set_ascii_mode(std::cin);
    std::cin >> p >> q;

    std::ifstream f("data.txt");
    CGAL::set_binary_mode(f);
    f >> s >> p;

    return 1;
}
```

79.3 IO for non-CGAL types

79.3.1 Using Output Formatting

To ensure that non-CGAL types are formatted correctly (i.e., respecting *IO::Mode*), *oformat* can be used. For types with a *Output_rep* specialization, the respective output routine of *Output_rep* will be called by *oformat*. Otherwise, the stream output operator will be called.

Example

```
std::cout << CGAL::oformat( myobject );
```

Optional, you can provide a second template parameter `F` as a formatting tag:

Example

```
std::cout << CGAL::oformat( myobject, My_formatting_tag() );
```

For a list of formatting tags supported by the type `T`, please refer to the documentation of the respective type.

79.3.2 Customizing Output Formatting

In some situations, you want to control the output formatting for a type `T`. For external types (third party libraries etc.), there might be problems if their stream output operator does not respect *IO::Mode*. The purpose of *Output_rep* is to provide a way to control output formatting that works independently of the object's stream output operator.

Instead of putting `T` directly into an output stream, `T` is wrapped into an output representation *Output_rep*. For convenience, a function *oformat* exists which constructs an instance of *Output_rep*.

If you do not specialize *Output_rep* for `T`, `T`'s stream output operator is called from within *Output_rep*, by default. If you want another behaviour for your type `T`, you have to provide a specialization for that type. Furthermore, you can provide specializations with a second template parameter (a formatting tag). The second template parameter defaults to *Null_tag* and means *default behaviour*.

For example, specializing *Output_rep* for *CORE::BigRat* (without a formatting tag parameter) could look like this:

Example

```
template <class F>
class Output_rep< ::CORE::BigRat, F> {
    const ::CORE::BigRat& t;
public:
    Output_rep( const ::CORE::BigRat& tt) : t(tt) {}

    std::ostream& operator()( std::ostream& out) const {
        switch (get_mode(out)) {
        case IO::PRETTY:{
            if(CGAL_CORE_DENOMINATOR(t) == ::CORE::BigRat(1))
                return out <<CGAL_CORE_NUMERATOR(t);
            else
                return out << CGAL_CORE_NUMERATOR(t)
                           << "/"
                           << CGAL_CORE_DENOMINATOR(t);
            break;
        }

        default:
            return out << CGAL_CORE_NUMERATOR(t)
                       << "/"
                       << CGAL_CORE_DENOMINATOR(t);
        }
    }
};
```

79.4 Colors

An object of the class *CGAL::Color* is a color available for drawing operations in many CGAL output streams.

Each color is defined by a triple of integers (r,g,b) with $0 \leq r,g,b \leq 255$, the so-called *rgb-value* of the color. There are a 11 predefined *Color* constants available: *BLACK*, *WHITE*, *GRAY*, *RED*, *GREEN*, *DEEPBLUE*, *BLUE*, *PURPLE*, *VIOLET*, *ORANGE*, and *YELLOW*.

79.5 Stream Support

Three classes are provided by CGAL as adaptors to input and output stream iterators. The class *Istream_iterator* is an input iterator adaptor and is particularly useful for classes that are similar but not compatible to *std::istream*. Similarly, the class *Ostream_iterator* is an output iterator adaptor. The class *Verbose_ostream* can be used as an output stream. The stream output operator << is defined for any type. The class stores in an internal state a stream and whether the output is active or not. If the state is active, the stream output operator << uses the internal stream to output its argument. If the state is inactive, nothing happens.

IO Streams
Reference Manual

Andreas Fabri, Geert-Jan Giezeman, and Lutz Kettner

All classes in the CGAL kernel provide input and output operators for IO streams. CGAL provides three different printing mode, defined in the enum *Mode*, as well as different functions to set and get the printing mode.

79.6 Classified Reference Pages

Enum

Functions

Classes

79.7 Alphabetical List of Reference Pages

CGAL::Color

Class

Definition

An object of the class *Color* is a color available for drawing operations in many CGAL output streams. Each color is defined by a triple of unsigned chars (r,g,b) with $0 \leq r,g,b \leq 255$, the so-called *rgb-value* of the color.

#include <CGAL/IO/Color.h>

Creation

Color c;	creates a color with rgb-value $(0,0,0)$, i.e. black.
Color c(unsigned char red, unsigned char green, unsigned char blue);	creates a color with rgb-value *(red,green,blue)*.

Operations

bool	$c == q$	Test for equality: Two colors are equal, iff their rgb-values are equal.
bool	$c\ != q$	Test for inequality.
unsigned char	*c.red() const*	returns the red component of c.
unsigned char	*c.green() const*	returns the green component of c.
unsigned char	*c.blue() const*	returns the blue component of c.

Constants

The following constants are predefined:

const Color	*BLACK = Color(0, 0, 0);*	Black.
const Color	*WHITE = Color(255, 255, 255);*	White.
const Color	*RED = Color(255, 0, 0);*	Red.
const Color	*GREEN = Color(0, 255, 0);*	Green.
const Color	*BLUE = Color(0, 0, 255);*	Blue.
const Color	*VIOLET = Color(255, 0, 255);*	Violet.
const Color	*ORANGE = Color(255, 170, 0);*	Orange.

See Also

CGAL::Qt_widget
CGAL::Geomview_stream

Function

CGAL::get_mode

Mode	*get_mode(std::ios& s)*	returns the printing mode of the IO stream *s*.

See Also

CGAL::Mode . . . . page 4424
CGAL::set_mode . . . . page 4433
CGAL::set_ascii_mode . . . . page 4431
CGAL::set_binary_mode . . . . page 4432
CGAL::set_pretty_mode . . . . page 4434
CGAL::is_ascii . . . . page 4420
CGAL::is_binary . . . . page 4421
CGAL::is_pretty . . . . page 4422

CGAL::operator>>

Definition

CGAL defines input operators for classes that are derived from the class *istream*. This allows to read from istreams as *cin*, as well as from strstreams and fstreams. The input operator is defined for all classes in the CGAL kernel.

istream&	*istream& is >> Class c*	Extracts object *c* from the stream *is*. Returns *is*.

See Also

Example

```
#include <CGAL/basic.h>
#include <iostream>
#include <fstream>

#include <CGAL/Cartesian.h>
#include <CGAL/Segment_2.h>

typedef CGAL::Point_2< CGAL::Cartesian<double> >     Point;
typedef CGAL::Segment_2< CGAL::Cartesian<double> >   Segment;

int
main()
{
    Point p, q;
    Segment s;

    CGAL::set_ascii_mode(std::cin);
    std::cin >> p >> q;

    std::ifstream f("data.txt");
    CGAL::set_binary_mode(f);
    f >> s >> p;

    return 1;
}
```

Function

CGAL::is_ascii

bool *is_ascii(std::ios& s)* checks if the IO stream *s* is in *ASCII* mode.

See Also

CGAL::Mode . . . . . . page 4424
CGAL::set_mode . . . . . . page 4433
CGAL::set_ascii_mode . . . . . . page 4431
CGAL::set_binary_mode . . . . . . page 4432
CGAL::set_pretty_mode . . . . . . page 4434
CGAL::get_mode . . . . . . page 4418
CGAL::is_binary . . . . . . page 4421
CGAL::is_pretty . . . . . . page 4422

CGAL::is_binary

bool	*is_binary(std::ios& s)*	checks if the IO stream *s* is in *BINARY* mode.

See Also

CGAL::Mode . . . . page 4424
CGAL::set_mode . . . . page 4433
CGAL::set_ascii_mode . . . . page 4431
CGAL::set_binary_mode . . . . page 4432
CGAL::set_pretty_mode . . . . page 4434
CGAL::get_mode . . . . page 4418
CGAL::is_ascii . . . . page 4420
CGAL::is_pretty . . . . page 4422

Function

Function

CGAL::is_pretty

bool	*is_pretty(std::ios& s)*	checks if the IO stream *s* is in *PRETTY* mode.

See Also

CGAL::Mode . . . . page 4424
CGAL::set_mode . . . . page 4433
CGAL::set_ascii_mode . . . . page 4431
CGAL::set_binary_mode . . . . page 4432
CGAL::set_pretty_mode . . . . page 4434
CGAL::get_mode . . . . page 4418
CGAL::is_ascii . . . . page 4420
CGAL::is_binary . . . . page 4421

CGAL::Istream_iterator<T,Stream>

Class

Definition

The class *Istream_iterator<T,Stream>* is an input iterator adaptor for the input stream class *Stream* and value type T. It is particularly useful for classes that are similar but not compatible to *std::istream*.

#include <CGAL/IO/Istream_iterator.h>

Creation

Istream_iterator<T,Stream> i;	creates an end-of-stream iterator *i*. This is a past-the-end iterator, and it is useful when constructing a range.
Istream_iterator<T,Stream> i(Stream& s);	creates an input iterator *i* reading from *s*. When *s* reaches end of stream, this iterator will compare equal to an end-of-stream iterator created using the default constructor.

Operations

i fulfills the requirements for an input iterator.

CGAL::Mode

Definition

All classes in the CGAL kernel provide input and output operators for IOStreams. The basic task of such an operator is to produce a representation of an object that can be written as a sequence of characters on devices as a console, a file, or a pipe. The enum *Mode* distinguish between three different printing formats.

In *ASCII* mode, numbers e.g. the coordinates of a point or the coefficients of a line, are written in a machine independent format. In BINARY mode, data are written in a binary format, e.g. a double is represented as a sequence of four byte. The format depends on the machine. The mode PRETTY serves mainly for debugging as the type of the geometric object is written, as well as the data defining the object. For example for a point at the origin with Cartesian double coordinates, the output would be *PointC2(0.0, 0.0)*. At the moment CGAL does not provide input operations for pretty printed data. By default a stream is in ASCII mode.

enum Mode { *ASCII = 0, BINARY, PRETTY* }*;*

See Also

CGAL::set_mode . . . . page 4433
CGAL::set_ascii_mode . . . . page 4431
CGAL::set_binary_mode . . . . page 4432
CGAL::set_pretty_mode . . . . page 4434
CGAL::get_mode . . . . page 4418
CGAL::is_ascii . . . . page 4420
CGAL::is_binary . . . . page 4421
CGAL::is_pretty . . . . page 4422

CGAL::Ostream_iterator<T,Stream>

Definition

The class *Ostream_iterator<T,Stream>* is an output iterator adaptor for the output stream class *Stream* and value type *T*.

#include <CGAL/IO/Ostream_iterator.h>

Creation

Ostream_iterator<T,Stream> o(Stream& s);	creates an output iterator *o* writing to *s*.

Operations

o fulfills the requirements for an output iterator.

Implementation

The *operator*()* in class *Ostream_iterator<T,Stream>* uses a proxy class.

Function

CGAL::operator<<

Definition

CGAL defines output operators for classes that are derived from the class *ostream*. This allows to write to ostreams as *cout* or *cerr*, as well as to strstreams and fstreams. The output operator is defined for all classes in the CGAL kernel and for the class *Color* as well. Let *os* be an output stream.

ostream& *ostream& os >> Class c*

Inserts object *c* in the stream *os*. Returns *os*.

See Also

CGAL::set_mode . . . page 4433
CGAL::set_ascii_mode . . . page 4431
CGAL::set_binary_mode . . . page 4432
CGAL::set_pretty_mode . . . page 4434
CGAL::get_mode . . . page 4418
CGAL::is_ascii . . . page 4420
CGAL::is_binary . . . page 4421
CGAL::is_pretty . . . page 4422
CGAL::operator>> . . . page 4419

Example

```
#include <CGAL/basic.h>
#include <iostream>
#include <fstream>

#include <CGAL/Cartesian.h>
#include <CGAL/Segment_2.h>

typedef CGAL::Point_2< CGAL::Cartesian<double> >     Point;
typedef CGAL::Segment_2< CGAL::Cartesian<double> >   Segment;

int main()
{
    Point p(0,1), q(2,2);
    Segment s(p,q);

    CGAL::set_pretty_mode(std::cout);
    std::cout << p << std::endl << q  << std::endl;

    std::ofstream f("data.txt");
    CGAL::set_binary_mode(f);
    f << s << p ;

    return 1;
}
```

CGAL::Input_rep<T,F>

Definition

The definition of *Input_rep* is completely symmetric to *Output_rep*.

Class

Class

CGAL::Output_rep<T,F>

Definition

The purpose of *Output_rep* is to provide a way to control output formatting that works independently of the object's stream output operator.

If you dont specialize *Output_rep* for `T`, `T`'s stream output operator is called from within *Output_rep*, by default. If you want another behaviour for your type `T`, you have to provide a specialization for that type. Furthermore, you can provide specializations with a second template parameter (a formatting tag). The second template parameter defaults to *Null_tag* and means DEFAULT BEHAVIOUR.

Specializations of *Output_rep* should provide the following features:

```
template< class F >
struct Output_rep< Some_type, F > {
    Output_rep( const Some_type& t );
    std::ostream& operator()( std::ostream& out ) const;
};
```

You can also specialize for a formatting tag `F`.

CGAL::iformat

The definition of the function *iformat* is completely symmetric to *oformat*.

Function

CGAL::oformat

Definition

Convenience function to construct an output representation (*Output_rep*) for type `T`.

template <class T>
Output_rep<T>

oformat(T t)

Generic IO for type *T*.

template <class T, typename F>
Output_rep<T,F>

oformat(T t, F)

Generic IO for type *T* with formatting tag.

CGAL::set_ascii_mode

Mode *set_ascii_mode(std::ios& s)*

sets the mode of the IO stream *s* to be the *ASCII* mode. Returns the previous mode of *s*.

Function

See Also

CGAL::Mode . . . . page 4424
CGAL::set_mode . . . . page 4433
CGAL::set_binary_mode . . . . page 4432
CGAL::set_pretty_mode . . . . page 4434
CGAL::get_mode . . . . page 4418
CGAL::is_ascii . . . . page 4420
CGAL::is_binary . . . . page 4421
CGAL::is_pretty . . . . page 4422

Function

CGAL::set_binary_mode

Mode *set_binary_mode(std::ios& s)*

sets the mode of the IO stream *s* to be the *BINARY* mode. Returns the previous mode of *s*.

See Also

CGAL::Mode ... page 4424
CGAL::set_mode ... page 4433
CGAL::set_ascii_mode ... page 4431
CGAL::set_pretty_mode ... page 4434
CGAL::get_mode ... page 4418
CGAL::is_ascii ... page 4420
CGAL::is_binary ... page 4421
CGAL::is_pretty ... page 4422

CGAL::set_mode

Mode *set_mode(std::ios& s, IO::Mode m)*

sets the printing mode of the IO stream *s*.

See Also

Function

Function

CGAL::set_pretty_mode

Mode *set_pretty_mode(std::ios& s)*

sets the mode of the IO stream *s* to be the *PRETTY* mode. Returns the previous mode of *s*.

See Also

CGAL::Verbose_ostream

Class

Definition

The class *Verbose_ostream* can be used as an output stream. The stream output operator << is defined for any type. The class *Verbose_ostream* stores in an internal state a stream and whether the output is active or not. If the state is active, the stream output operator << uses the internal stream to output its argument. If the state is inactive, nothing happens.

#include <*CGAL/IO/Verbose_ostream.h*>

Creation

Verbose_ostream verr(bool active = false, std::ostream& out = std::cerr);

creates an output stream with state set to *active* that writes to the stream *out*.

Operations

template < class T >
Verbose_ostream& *verr << T t*

Example

The class *Verbose_ostream* can be conveniently used to implement for example the *is_valid()* member function for triangulations or other complex data structures.

```
bool is_valid( bool verbose = false, int level = 0) {
    Verbose_ostream verr( verbose);
    verr << "Triangulation::is_valid( level = " << level << ')' << endl;
    verr << "    Number of vertices = " << size_of_vertices() << endl;
    // ...
}
```

Part XVIII

Visualization

Chapter 80

Geomview

Andreas Fabri and Sylvain Pion

Contents

80.1 Definition

This chapter presents the CGAL interface to Geomview[1], which is a viewer for three-dimensional objects, originally developed at the Geometry Center in Minneapolis[2].

Note: The functionality described in this chapter is not available on Windows.

An object of the class *Geomview_stream* is a stream in which geometric objects can be inserted and where geometric objects can be extracted from. The constructor starts Geomview either on the local either on a remote machine.

Not all but most classes of the CGAL kernel have output operators for the *Geomview_stream*. 2D objects are embedded in the *xy*-plane. Input is only provided for points. Polyhedron and 2D and 3D triangulations have output operators for the *Geomview_stream*.

80.2 Implementation

The constructor forks a process and establishes two pipes between the processes. The forked process is then overlaid with Geomview. The file descriptors *stdin* and *stdout* of Geomview are hooked on the two pipes.

[1] `http://www.geomview.org/`
[2] `http://www.geom.umn.edu/`

All insert operators construct expressions in *gcl*, the Geomview command language, which is a subset of LISP. These expressions are sent to Geomview via the pipe. The extract operators notify *interest* for a certain kind of events. When such an event happens Geomview sends a description of the event in *gcl* and the extract operator has to parse this expression.

In order to implement further insert and extract operators you should take a look at the implementation and at the Geomview manual.

80.3 Example

The following program ouputs successively a 2D Delaunay triangulation (projected), a 3D Delaunay, and a terrain from the set of points.

```
#include <CGAL/Cartesian.h>
#include <iostream>

#ifndef CGAL_USE_GEOMVIEW
int main()
{
  std::cout << "Geomview doesn't work on Windows, so..." << std::endl;
  return 0;
}
#else

#include <fstream>
#include <unistd.h> // for sleep()

#include <CGAL/Projection_traits_xy_3.h>

#include <CGAL/Delaunay_triangulation_2.h>
#include <CGAL/Delaunay_triangulation_3.h>

#include <CGAL/IO/Geomview_stream.h>
#include <CGAL/IO/Triangulation_geomview_ostream_2.h>
#include <CGAL/IO/Triangulation_geomview_ostream_3.h>

#include <CGAL/intersections.h>

typedef CGAL::Cartesian<double>  K;

typedef K::Point_2 Point2;
typedef CGAL::Projection_traits_xy_3<K> Gt3;
typedef Gt3::Point Point3;

typedef CGAL::Delaunay_triangulation_2<K>   Delaunay;
typedef CGAL::Delaunay_triangulation_2<Gt3> Terrain;

typedef CGAL::Delaunay_triangulation_3<K>   Delaunay3d;

int main()
{
```

```
  CGAL::Geomview_stream gv(CGAL::Bbox_3(-100, -100, -100, 600, 600,
    600));
  gv.set_line_width(4);
  // gv.set_trace(true);
  gv.set_bg_color(CGAL::Color(0, 200, 200));
  // gv.clear();

  Delaunay D;
  Delaunay3d D3d;
  Terrain T;
  std::ifstream iFile("data/points3", std::ios::in);
  Point3 p;

  while ( iFile >> p )
  {
      D.insert( Point2(p.x(), p.y()) );
      D3d.insert( p );
      T.insert( p );
  }

  // use different colors, and put a few sleeps/clear.

  gv << CGAL::BLUE;
  std::cout << "Drawing 2D Delaunay triangulation in wired mode.\n";
  gv.set_wired(true);
  gv << D;

#if 1 // It's too slow !  Needs to use OFF for that.
  gv << CGAL::RED;
  std::cout << "Drawing its Voronoi diagram.\n";
  gv.set_wired(true);
  D.draw_dual(gv);
#endif

  sleep(5);
  gv.clear();

  std::cout << "Drawing 2D Delaunay triangulation in non-wired mode.\n";
  gv.set_wired(false);
  gv << D;
  sleep(5);
  gv.clear();

  std::cout << "Drawing 3D Delaunay triangulation in wired mode.\n";
  gv.set_wired(true);
  gv << D3d;
  sleep(5);
  gv.clear();
  std::cout << "Drawing 3D Delaunay triangulation in non-wired mode.\n";
  gv.set_wired(false);
  gv << D3d;
  sleep(5);
  gv.clear();
```

```
  std::cout << "Drawing Terrain in wired mode.\n";
  gv.set_wired(true);
  gv << T;
  sleep(5);
  gv.clear();
  std::cout << "Drawing Terrain in non-wired mode.\n";
  gv.set_wired(false);
  gv << T;

  std::cout << "Enter a key to finish" << std::endl;
  char ch;
  std::cin >> ch;

  return 0;
}
#endif
```

File: demo/Geomview/gv_terrain.cpp

Geomview
Reference Manual

Andreas Fabri and Sylvain Pion

This chapter presents the CGAL interface to Geomview[3], which is a viewer for three-dimensional objects, originally developed at the Geometry Center in Minneapolis[4].

Geomview 1.8.1 is required.

Classes

80.4 Alphabetical List of Reference Pages

[3]`http://www.geomview.org/`
[4]`http://www.geom.umn.edu/`

Class

CGAL::Geomview_stream

Definition

An object of the class *Geomview_stream* is a stream in which geometric objects can be inserted and where geometric objects can be extracted from. The constructor starts Geomview either on the local either on a remote machine.

#include <CGAL/IO/Geomview_stream.h>

Creation

Geomview_stream gs(Bbox_3 bbox = Bbox_3(0,0,0, 1,1,1),
*const char *machine = NULL,*
*const char *login = NULL)*

Introduces a Geomview stream *gs* with a camera that sees the bounding box. The command *geomview* must be in the user's *PATH*. If *machine* and *login* are not *NULL*, Geomview is started on the remote machine using *rsh*.

Operations

Output Operators for CGAL Kernel Classes

At the moment not all classes of the CGAL kernel have output operators. 2D objects are embedded in the *xy*-plane.

template <class R>
Geomview_stream& *Geomview_stream& G << Point_2<R> p*

Inserts the point *p* into the stream *gs*.

template <class R>
Geomview_stream& *Geomview_stream& G << Point_3<R> p*

Inserts the point *p* into the stream *gs*.

template <class R>
Geomview_stream& *Geomview_stream& G << Segment_2<R> s*

Inserts the segment *s* into the stream *gs*.

template <class R>
Geomview_stream& *Geomview_stream& G << Segment_3<R> s*

Inserts the segment *s* into the stream *gs*.

template <class R>
Geomview_stream& *Geomview_stream& G << Ray_2<R> r*

Inserts the ray *r* into the stream *gs*.

template <class R>
Geomview_stream& *Geomview_stream& G << Ray_3<R> r*

Inserts the ray *r* into the stream *gs*.

template <class R>
Geomview_stream& *Geomview_stream& G << Line_2<R> l*

Inserts the line *l* into the stream *gs*.

template <class R>
Geomview_stream& *Geomview_stream& G << Line_3<R> l*

Inserts the line *l* into the stream *gs*.

template <class R>
Geomview_stream& *Geomview_stream& G << Triangle_2<R> t*

Inserts the triangle *t* into the stream *gs*.

template <class R>
Geomview_stream& *Geomview_stream& G << Triangle_3<R> t*

Inserts the triangle *t* into the stream *gs*.

template <class R>

Geomview_stream& *Geomview_stream& G << Tetrahedron_3<R> t*

Inserts the tetrahedron *t* into the stream *gs*.

template <class R>
Geomview_stream& *Geomview_stream& G << Sphere_3<R> s*

Inserts the sphere *s* into the stream *gs*.

Geomview_stream& *Geomview_stream& G << Bbox_2 b*

Inserts the bounding box *b* into the stream *gs*.

Geomview_stream& *Geomview_stream& G << Bbox_3 b*

Inserts the bounding box *b* into the stream *gs*.

template < class InputIterator >
void *gs.draw_triangles(InputIterator begin, InputIterator end)*

[*begin*;*end*) is an iterator range with value type *Triangle_3< R>*. This method uses the OFF format to draw several triangles at once, which is much faster than drawing them one by one.

Input Operators for CGAL Kernel Classes

At the moment input is only provided for points. The user has to select a point on the *pick plane* with the right mouse button. The pick plane can be moved anywhere with the left mouse button, before a point is entered.

template <class R>
Geomview_stream& *Geomview_stream& G >> Point_3<R>& p*

Extracts the point *p* from the stream *gs*. The point is echoed by default, and it depends on the stream echo mode status.

Output Operators for CGAL Basic Library Classes

#include <CGAL/IO/Polyhedron_geomview_ostream.h>

template <class Traits, class HDS>
Geomview_stream& *Geomview_stream &G << Polyhedron_3<Traits,HDS> P*

Inserts the polyhedron *P* into the stream *gs*.

#include <CGAL/IO/Triangulation_geomview_ostream_2.h>

template <class GT, class TDS>

Geomview_stream&	*Geomview_stream &G << Triangulation_2<GT,TDS> T*	
		Inserts the 2D triangulation *T* into the stream *gs*. The actual output depends on whether the stream is in wired mode or not. Also note that in the case of terrains (when *GT::Point_2* is *Point_3<R>*), then the 3D terrain is displayed.

#include <CGAL/IO/Triangulation_geomview_ostream_3.h>

template <class GT, class TDS>

Geomview_stream&	*Geomview_stream &G << Triangulation_3<GT,TDS> T*	
		Inserts the 3D triangulation *T* into the stream *gs*. The actual output depends on whether the stream is in wired mode or not.

Colors

Geomview distinguishes between edge and face colors. The edge color is at the same time the color of vertices.

Geomview_stream&	*gs << Color c*	Makes *c* the color of vertices, edges and faces in subsequent IO operations.
Color	*gs.set_bg_color(Color c)*	
		Changes the background color. Returns the old value.
Color	*gs.set_vertex_color(Color c)*	
		Changes the vertex color. Returns the old value.
Color	*gs.set_edge_color(Color c)*	
		Changes the edge color. Returns the old value.
Color	*gs.set_face_color(Color c)*	
		Changes the face color. Returns the old value.

Miscellaneous

void	*gs.clear()*	Deletes all objects.
void	*gs.pickplane()*	Creates a pickplane (useful after a clear).
void	*gs.look_recenter()*	Positions the camera in a way that all objects can be seen.

int	*gs.get_line_width() const*	
		Returns the line width.
int	*gs.set_line_width(int w)*	
		Sets the line width to w. Returns the previous value.
double	*gs.get_vertex_radius() const*	
		Returns the radius of vertices.
double	*gs.set_vertex_radius(double r) const*	
		Sets the radius of vertices to d. Returns the previous value.
string	*gs.get_new_id(string s)*	
		Used to obtain unique identifier names passed to Geomview. On successive calls with the same s value, it will return a string which is s appended with the numbers 0, then 1, then 2... Note that all counters are reset when *clear*() is called.
bool	*gs.set_wired(bool b)*	Sets wired mode. In wired mode, some structures output only there edges, not there surfaces. Returns the previous value. By default, wired mode is off.
bool	*gs.get_wired()*	Returns *true* iff wired mode is on.

advanced

Advanced and Developers Features

The following functions are helpful if you develop your own insert and extract functions. The following functions allow to pass a string from Geomview and to read data sent back by Geomview.

Geomview_stream&	*gs << string s*	Inserts string s into the stream.
Geomview_stream&	*gs >> char* s*	Extracts a string s from the stream. *Precondition*: You have to allocate enough memory.
Geomview_stream&	*gs << int i*	Inserts i into the stream. Puts whitespace around if the stream is in ascii mode.
Geomview_stream&	*gs << unsigned int i*	Inserts i into the stream. Puts whitespace around if the stream is in ascii mode.

Geomview_stream&	*gs << long i*	Inserts *i* into the stream. Puts whitespace around if the stream is in ascii mode. Currently implemented by converting to int, so it can be truncated on 64 bit platforms.
Geomview_stream&	*gs << unsigned long i*	Inserts *i* into the stream. Puts whitespace around if the stream is in ascii mode. Currently implemented by converting to unsigned int, so it can be truncated on 64 bit platforms.
Geomview_stream&	*gs << double d*	Inserts double *d* into the stream. Puts whitespace around if the stream is in ascii mode.
bool	*gs.set_trace(bool b)*	Sets tracing on. The data that are sent to *Geomview* are also sent to *cerr*. Returns the previous value. By default tracing is off.
bool	*gs.get_trace()*	Returns *true* iff tracing is on.
bool	*gs.set_raw(bool b)*	Sets raw mode. In raw mode, kernel points are output without headers and footers, just the coordinates (in binary or ascii mode). This allows the implementation of the stream functions for other objects to re-use the code for points internally, by temporary saving the raw mode to true, and restoring it after. Returns the previous value. By default, raw mode is off.
bool	*gs.get_raw()*	Returns *true* iff raw mode is on.
bool	*gs.set_echo(bool b)*	Sets echo mode. In echo mode, when you select a point in Geomview, the point is actually sent back to Geomview. Returns the previous value. By default, echo mode is on.
bool	*gs.get_echo()*	Returns *true* iff echo mode is on.
bool	*gs.set_binary_mode(bool b = true)*	Sets whether we are in binary mode.
bool	*gs.set_ascii_mode(bool b = true)*	Sets whether we are in ascii mode.
bool	*gs.get_binary_mode()*	Returns *true* iff *gs* is in binary mode.
bool	*gs.get_ascii_mode()*	Returns *true* iff *gs* is in ascii mode.

advanced

Implementation

The constructor forks a process and establishes two pipes between the processes. The forked process is then overlaid with Geomview. The file descriptors *stdin* and *stdout* of Geomview are hooked on the two pipes.

All insert operators construct expressions in *gcl*, the Geomview command language, which is a subset of LISP. These expressions are sent to Geomview via the pipe. The extract operators notify *interest* for a certain kind of events. When such an event happens Geomview sends a description of the event in *gcl* and the extract operator has to parse this expression.

In order to implement further insert and extract operators you should take a look at the implementation and at the Geomview manual.

Chapter 81

CGAL and the Qt Graphics View Framework

Andreas Fabri and Laurent Rineau

Qt is a GUI toolkit for cross-platform application development.

81.1 Introduction

This chapter describes classes that help to visualize two dimensional CGAL objects with the Qt Graphics View Framework.

This framework uses the model view paradigm. *QGraphicsItem*s are stored in a *QGraphicsScene* and are displayed in a *QGraphicsView*. The items have a paint method which is called when an item is in the visible area of a view. The framework is also responsible for dispatching events from the view via the scene to the items. The framework is extensible in the sense that users can add classes derived from *QGraphicsItem*.

Besides visualizing *QGraphicsItem*s users want to enter geometric objects. We provide the input generators for all 2D CGAL kernel objects.

The package includes also a class for providing zooming, panning, and scrolling to the graphics view.

The following sections describe the interaction between all these classes, We finally describe the internals of a *QGraphicsItem*.

81.1.1 Naming Conventions

As Qt and CGAL have different naming conventions, and as this package brings them together we adopted the following, hybrid naming conventions.

- All header files are in the directory *CGAL/Qt/*.
- Class names are concatenated capitalized words, and function names are concatenated capitalized word with the first word in lowercase. The rationale is that these classes are related to Qt, and that they sometimes are derived classes that have to override member functions adhering to this naming scheme.

- All classes are in the nested namespace *CGAL::Qt*.

81.2 Overall Design

In Figure 81.1 you see four classes depicted in grey, that come from the Qt Graphics View Framework. The *QGraphicsScene* contains *QGraphicsItem*s, which get displayed in any number of *QGraphicsView*s. The views are widgets, that is they take screen space in an application.

The fourth class is the *QObject*. It plays an important role in Qt for event handling and memory management. First, it allows to add signals and slots, and to connect them. Second, it allows to install event filters.

81.2.1 Visualizing CGAL Datastructures

In order to visualize for example a *CGAL::Delaunay_triangulation_2<K>*, we provide the graphics item class *CGAL::Qt::TriangulationGraphicsItem<T>*. It provides a *paint* method that draws the edges and vertices of a triangulation using the drawing primitives of the *QPainter*. The color of vertices and edges, can be chosen by setting a user defined *QPen*.

As this graphics item only stores a pointer to a triangulation, it must be notified about changes like the insertion of points coming from a file, a process or as input generated with the mouse. We use the signal/slot mechanism of Qt for that purpose, that is when the triangulation changes the application emits a signal that can get connected to the *modelChanged()* slot of the graphics item.

81.2.2 Navigation

We provide a class *CGAL::Qt::GraphicsViewNavigation* that can be installed as an event filter of a graphics view and its viewport. As for all Qt widgets, the *QGraphicsView* is derived from *QObject*. Events like the mouse buttons pressed or released, the mouse movemed, keys pressed, are passed to the view which first hands them over to the event filter. The *CGAL::Qt::GraphicsViewNavigation* event filter allows to zoom, scroll, and recenter. Finally, the class emits a signal with the mouse coordinates. This can be used to display the current mouse position in the status bar of an application.

81.2.3 Generation of Input

Input of CGAL kernel objects, polylines, etc. is generated by classes derived from *CGAL::Qt::GraphicsViewInput*. As the navigation class, they are event handler of the graphics view, because they have to know where the mouse is, when the user clicks in order to enter a point.

Once the input generator has assembled the object, which can involve several mouse clicks, it emits a *CGAL::Object* wrapping the input. The emitted input can be connected by the application developer to a slot. In the 2D demos of CGAL, which use the Graphics View Framework we connect it to the slot *MainWindow::processInput(CGAL::Object)*. This method unwraps from the *CGAL::Object* and inserts it in the data structure. It then typically emits a signal *modelChanged()* which can be connected to the graphics item representing the data structure.

All input generators we provide use the left mouse button for entering points. The right click terminates a sequence of entered points. 'Delete' and 'backspace' remove the last entered point. 'Esc' resets the input generator. As the 'Ctrl' key is used by the *CGAL::Qt::GraphicsViewNavigation* this modifier is not used.

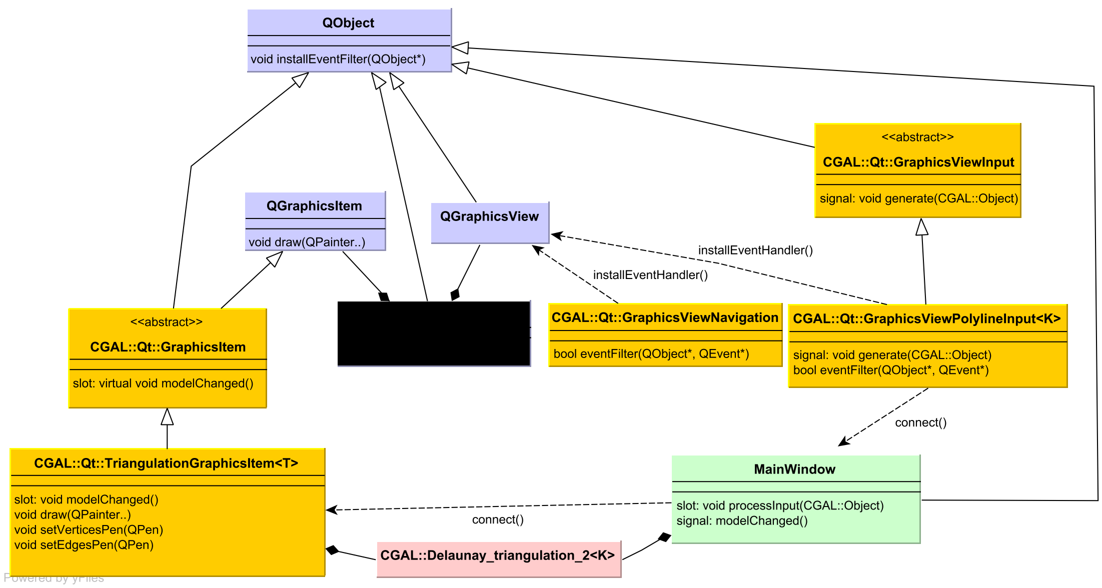


Figure 81.1: UML Class Diagram with the Qt classes (blue), CGAL classes for using the framework (yellow), CGAL data structures (red), and application classes (green).

CGAL and the Qt Graphics View Framework
Reference Manual

Andreas Fabri and Laurent Rineau

This package provides some classes which allow to use CGAL classes in *Qt* applications which make use of the Qt Graphics View Framework.

81.3 Classified Reference Pages

GraphicsItem Classes

Input Classes

Miscellaneous Classes

81.4 Alphabetical List of Reference Pages

CGAL::Qt::Converter<K>

Definition

An object of type *Converter<K>* converts 2D CGAL kernel objects to their equivalent objects in *Qt*, and the other way round. Note that some objects have no equivalent. For example the *CGAL::Circle_2<K>* cannot be converted to something in Qt, and the unbounded objects *CGAL::Ray_2<K>* and *CGAL::Line_2<K>* are clipped. Note also that CGAL and *Qt* sometimes also use the same word for different things. For example *line* denotes an unbounded line in CGAL, wheras it denotes a bounded segment in *Qt*.

#include <CGAL/Qt/Converter.h>

Creation

Converter<K> c(QRectF clippingRect);	The clipping rect is used for unbounded CGAL objects.

From CGAL to Qt

QRectF	*c(Bbox_2)*	Converts a bounding box.
QPointF	*c(K::Point_2)*	Converts a point.
QLineF	*c(K::Segment_2)*	Converts a segment.
QLineF	*c(K::Ray_2)*	Clips the ray and converts the resulting segment.
QLineF	*c(K::Line_2)*	Clips the line and converts the resulting segment.
QPolygonF	*c(K::Triangle_2)*	Converts a triangle.
QRectF	*c(K::Iso_rectangle_2)*	Converts an iso rectangle.
QPolygonF	*c(std::list<K::Point_2>)*	Converts a list of points to a polygon.

From Qt to CGAL

K::Point_2	*c(QPointF)*	Converts a point.
K::Segment_2	*c(QLineF)*	Converts a segment.
K::Iso_rectangle_2	*c(QRectF)*	Converts an iso rectangle.
std::list<K::Point_2>	*c(QPolygonF)*	Converts a polygon to a list of points.

Class

CGAL::Qt::PainterOstream<K>

Definition

An object of type *PainterOstream<K>* provides output operators for CGAL kernel objects. As CGAL has unbounded objects the *PainterOstream<K>* must be constructed with a clipping rectangle. This is typically the visible area of the widget where the unbounded object is displayed.

#include <CGAL/Qt/PainterOstream.h>

Creation

PainterOstream<K> os(QPainter qp, QRectF clippingRect);*

The clipping rect is used for unbounded CGAL objects.

Operations

PainterOstream<K>	*os << K::Point_2*	Draws a point.
PainterOstream<K>	*os << K::Segment_2*	Draws a segment.
PainterOstream<K>	*os << K::Ray_2*	Draws a clipped ray.
PainterOstream<K>	*os << K::Line_2*	Draws a clipped line.
PainterOstream<K>	*os << K::Triangle_2*	Draws a triangle.
PainterOstream<K>	*os << K::Iso_rectangle_2*	Draws an iso rectangle.
PainterOstream<K>	*os << K::Circle_2*	Draws a circle.
PainterOstream<K>	*os << K::Circular_arc_2*	Draws a circular arc.
PainterOstream<K>	*os << std::list<K::Point_2>*	Draws a polyline. In order to close it the first and last point must be equal.

PainterOstream<K>	*os << Bbox_2*	Draws an iso rectangle.
PainterOstream<K>	*os << QPen*	Sets the pen used in the next paint operations.
PainterOstream<K>	*os << QBrush*	Sets the brush used in the next paint operations.

Class

CGAL::Qt::GraphicsViewInput

Definition

An object of type *GraphicsViewInput* can emit a signal with *CGAL::Object* as argument.

#include <CGAL/Qt/GraphicsViewInput.h>

Signals

void	*in.generate(Object)*	A signal that emits a *CGAL::Object*.

CGAL::Qt::GraphicsViewPolylineInput<K>

Class

Definition

An object of type *GraphicsViewPolylineInput<K>* creates a list of points. A new vertex is inserted every time the left mouse button is pressed. The list of points is emitted on a right click or when the number of points specified in the constructor is reached. You can use the 'Del' or 'Backspace' key if you want to remove your last entered point in the polygon, and the 'Esc' key if you want to remove all points.

The tool can serve at the same time for entering a single point, a polyline with a given number of points, and for open as well as closed polylines.

For polylines the segment between the last entered point and the current mouse position is only drawn correctly when mouse tracking is enabled in the graphics view. The same holds for closed polygons.

#include <CGAL/Qt/GraphicsViewPolylineInput.h>

Parameters

The template parameter of *GraphicsViewPolylineInput<K>* must be a CGAL *Kernel*.

Inherits From

Qt::GraphicsViewInput

Creation

*GraphicsViewPolylineInput<K> in(QObject *p, QGraphicsScene* s, int n = 0, bool closed = true);*

p is a parent object. s is the scene where the polyline is generated. n is the number of points of the polyline to be generated. If c is equal to 0, the default value, the number of points of the polyline is not limited. When *closed* is *true* the function displays a polygon.

Signals

void *in.generate(CGAL::Object o)*

The object o contains a *std:list<K::Point_2>*.

Class

CGAL::Qt::GraphicsViewIsoRectangleInput<K>

Definition

An object of type *GraphicsViewIsoRectangleInput<K>* creates an axis parallel rectangle.

#include <CGAL/Qt/GraphicsViewIsoRectangleInput.h>

Parameters

The template parameter of *GraphicsViewIsoRectangleInput<K>* must be a CGAL *Kernel*.

Inherits From

GraphicsViewInput

Creation

*GraphicsViewIsoRectangleInput<K> in(QObject *p, QGraphicsScene* s);*

p is a parent object. s is the scene where the iso rectangle is generated.

Signals

void *in.generate(Object o)*

The object o contains a *K::Iso_rectangle_2*.

CGAL::Qt::GraphicsViewCircleInput<K>

Class

Definition

An object of type *GraphicsViewCircleInput<K>* creates a circle, defined by either center and radius, or two or three points on the circle. A new vertex is inserted every time the left mouse button is released. The *Del* key removes the last entered point. The *Esc* key removes all entered points.

#include <CGAL/Qt/GraphicsViewCircleInput.h>

Parameters

The template parameter of *GraphicsViewCircleInput<K>* must be a CGAL *Kernel*.

Inherits From

Qt::GraphicsViewInput

Creation

*GraphicsViewCircleInput<K> in(QObject *p, QGraphicsScene* s, int pointsOnCircle = 1);*

p is a parent object. *s* is the scene where the circle is generated. *pointsOnCircle* is the number of points on the circle to be generated, that is the default value *1* corresponds to the case center/radius.

Signals

void *in.generate(Object o)*

The object *o* contains a *std::pair<K::Point_2, K::FT>* for center and radius, or a *std::pair<K::Point_2, K::Point_2>* for two points defining the circle, or *CGAL::array<K::Point_2, 3>* for three points defining the circle.

Class

CGAL::Qt::GraphicsViewCircularArcInput<K>

Definition

An object of type *GraphicsViewCircularArcInput<K>* creates a circular arc, defined by three points on a circle. A new vertex is inserted every time the left mouse button is released. The *Del* key removes the last entered point. The *Esc* key removes all entered points.

#include <CGAL/Qt/GraphicsViewCircularArcInput.h>

Parameters

The template parameter of *GraphicsViewCircularArcInput<K>* must be a CGAL *CircularKernel*.

Inherits From

Qt::GraphicsViewInput

Creation

*GraphicsViewCircularArcInput<K> in(QObject *p, QGraphicsScene* s);*

p is a parent object. s is the scene where the circular arc is generated.

Signals

void *in.generate(Object o)*

The object o contains a *Circular_arc_2<K>*.

CGAL::Qt::GraphicsItem

Class

Definition

The CGAL graphics items hold pointers to CGAL datastructures. When a datastructure changes, a signal is emitted. An object derived from type *GraphicsItem* must provide an implementation of the virtual slot *modelChanged()*. This typically triggers redrawing, as well as recomputation of the bounding box which in turn may trigger redrawing of overlapping graphics items.

#include <CGAL/Qt/GraphicsItem.h>

Inherits From

QGraphicsItem

Slots

virtual void	*gi.modelChanged()*	This slot must be provided by derived classes.

Class

CGAL::Qt::CircularArcGraphicsItem<CK>

Definition

An object of type *CircularArcGraphicsItem<CK>* is a graphics item that encapsulates a circular arc.

#include <CGAL/Qt/CircularArcGraphicsItem.h>

Parameters

The template parameter of *CircularArcGraphicsItem<CK>* must be a CGAL *CircularKernel*.

Inherits From

Qt::GraphicsItem

Creation

CircularArcGraphicsItem<CK> cagi;

Constructs a graphics item for a circular arc.

Operations

QPen *cagi.edgesPen() const*

Returns the pen used to draw edges.

void *cagi.setEdgesPen()* Sets the pen used to draw edges.

void *cagi.setArc(Circular_arc_2<CK> ca2)*

Sets the circular arc.

CGAL::Qt::TriangulationGraphicsItem<T>

Class

Definition

An object of type *TriangulationGraphicsItem<T>* is a graphics item that encapsulates a 2D triangulation.

#include <CGAL/Qt/TriangulationGraphicsItem.h>

Parameters

The template parameter of *TriangulationGraphicsItem<T>* must be a CGAL 2D triangulation class.

Inherits From

Qt::GraphicsItem

Creation

TriangulationGraphicsItem<T> tgi(T t);*

Constructs a graphics item for a triangulation pointed by *t*.

Operations

QPen	*tgi.verticesPen() const*	Returns the pen used to draw vertices.
void	*tgi.setVerticesPen()*	Sets the pen used to draw vertices.
QPen	*tgi.edgesPen() const*	Returns the pen used to draw edges.
void	*tgi.setEdgesPen()*	Sets the pen used to draw edges.
bool	*tgi.visibleVertices()*	Returns *true*, iff vertices are drawn.
void	*tgi.setVisibleVertices(bool b)*	Sets the property.
bool	*tgi.visibleEdges()*	Returns *true*, iff edges are drawn.
void	*tgi.setVisibleEdges(bool b)*	Sets the property.

Class

CGAL::Qt::ConstrainedTriangulationGraphicsItem<CT>

Definition

An object of type *ConstrainedTriangulationGraphicsItem<CT>* is a graphics item that encapsulates a constrained triangulation.

#include <CGAL/Qt/ConstrainedTriangulationGraphicsItem.h>

Parameters

The template parameter of *ConstrainedTriangulationGraphicsItem<CT>* must be a CGAL 2D constrained triangulation class.

Inherits From

Qt::TriangulationGraphicsItem

Creation

ConstrainedTriangulationGraphicsItem<CT> ctgi(CT ct);*

Constructs a graphics item for triangulation *ct*.

Operations

QPen *ctgi.constraintsPen() const*

Returns the pen used to draw constraints.

void *ctgi.setConstraintsPen()*

Sets the pen used to draw constraints.

CGAL::Qt::VoronoiGraphicsItem<DT>

Class

Definition

An object of type *VoronoiGraphicsItem<DT>* is a graphics item that encapsulates a Delaunay triangulation in order to draw its dual, the Voronoi diagram.

#include <CGAL/Qt/VoronoiGraphicsItem.h>

Parameters

The template parameter *DT* of *VoronoiGraphicsItem<DT>* must be a CGAL 2D Delaunay triangulation class.

Inherits From

Qt::GraphicsItem

Creation

VoronoiGraphicsItem<DT> vgi(DT dt);*

Constructs a graphics item for the dual of the Delaunay triangulation *dt*.

Operations

QPen	*vgi.edgesPen() const*	Returns the pen used to draw edges.
void	*vgi.setEdgesPen()*	Set the pen used to draw edges.
bool	*vgi.visibleEdges()*	Returns *true*, iff edges are drawn.
bool	*vgi.setVisibleEdges(bool b)*	Set the property.

Class

CGAL::Qt::GraphicsViewNavigation

Definition

An object of type *GraphicsViewNavigation* can be added as event filter to a *Qt::QGraphicsView* and its viewport.

Dragging the left mouse button while holding the 'Ctrl' key defines a zoom rectangle. Dragging the right mouse button while holding the 'Ctrl' key translates the scene. 'Ctrl-Shift' and a click of the right mouse button translates what is under the mouse to the center.

#include <CGAL/Qt/GraphicsViewNavigation.h>

Operations

bool	*nav.eventFilter(QObject *obj, QEvent *event)*
	The event filter.

Signals

void	*nav.mouseCoordinates(QPointF p)*
	Emits the real world mouse coordinates.

Chapter 82

CGAL Ipelets

Sébastien Loriot and Sylvain Pion

Contents

82.1 Introduction

The Ipe extensible drawing editor (http://tclab.kaist.ac.kr/ipe/) [Sch95, Che09] is a tool used by computational geometry researchers to produce 2D figures for inclusion in articles or presentations. The extensible adjective sheds a light on an important feature: the possibility for users to write small extensions (called *ipelets*) to interface any algorithm with Ipe. This feature is especially interesting to produce illustrations in a paper describing a new algorithm. This package provides one class that eases the writing of ipelets based on CGAL, reducing the needed knowledge of the API of Ipe. This class is designed to fulfill most of the needs to implement an ipelet for a 2D CGAL algorithm. In addition, this package comes with more than ten complete examples of ipelets: alpha-shape, arrangements, Voronoi diagrams, convex hulls, Hilbert curve, k-order Delaunay, 2D mesh, Minkowski sum, polygon partition, random generators, triangulations . . .

82.2 CGAL Ipelets

The class *CGAL::Ipelet_base* derives from the class *Ipelets* from Ipe and has one template parameter indicating which kernel must be used within the ipelet. In practice, we recommend to use either *CGAL::Exact_predicate_exact_constructions_kernel* or *CGAL::Exact_predicate_inexact_construction_kernel*.

Two main methods are provided by the *CGAL::Ipelet_base* class. The first one, *read_active_objects* retrieves all primitives selected in Ipe when calling an ipelet, and converts them into equivalent CGAL objects. The second method, *draw_in_ipe* draws CGAL objects in the Ipe window. These functions handle basic objects such as points, segments, polygons, circles and circular arcs.

The registration of an ipelet into Ipe can be done using the macro *CGAL_IPELET*. *CGAL_IPELET(ipelet_class_name)* must follow the definition of the ipelet class, in the same file source file.

82.3 Example

The following example shows how the class *CGAL::Ipelet_base* can be used to easily interface the CGAL 2D Delaunay triangulation with Ipe.

```
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/Delaunay_triangulation_2.h>
#include <CGAL/CGAL_Ipelet_base.h>

namespace my_triangulation{

typedef CGAL::Exact_predicates_inexact_constructions_kernel
     Kernel;
typedef CGAL::Delaunay_triangulation_2<Kernel>
     Delaunay;

//Function names of the ipelet
const std::string labels[] = {  "Delaunay","Help" };
//Help message associated to the first function
const std::string hmsg[] = {
  "Draw a Delaunay triangulation of a set of points"
};

class Triangulation_ipelet
  : public CGAL::Ipelet_base<Kernel,2>{
public:
  //declare an ipelet called CGAL Delaunay, with 2 functions (including
   help message).
  Triangulation_ipelet()
    :CGAL::Ipelet_base<Kernel,2>("CGAL Delaunay",labels,hmsg){}
  void protected_run(int);
};

//function called when using the ipelet.
void Triangulation_ipelet::protected_run(int fn)
{
  switch (fn){
    case 1:
      show_help();//print an help message
      return;
    default:
    std::list<Point_2> pt_lst;

    //Recovering points using output iterator of type
    //Dispatch_or_drop_output_iterator
    read_active_objects(
      CGAL::dispatch_or_drop_output<Point_2>(std::back_inserter(pt_lst))
```

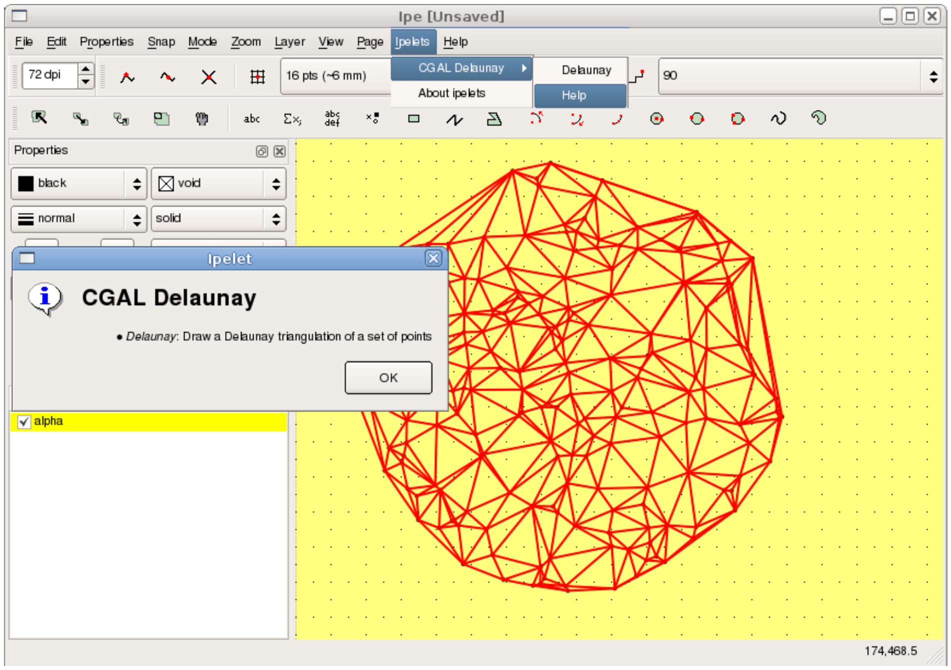


Figure 82.1: The ipelet of the example in action.

```
    );

    if (pt_lst.empty()) {
      print_error_message("No mark selected");
      return;
    }

    Delaunay dt;
    dt.insert(pt_lst.begin(),pt_lst.end());

    //draw the triangulation.
    draw_in_ipe(dt);
  };
}

}//namespace my_triangulation

//register the ipelet in Ipe
CGAL_IPELET(my_triangulation::Triangulation_ipelet)
```

File: demo/CGAL_ipelets/simple_triangulation.cpp

82.4 Installation of the Demo Ipelets

Some working ipelets can be found within the demo directory of this package. To install them you need to go to the demo directory of the package and run cmake. The compilation of these demo requires the source tree and the library of Ipe. If your Ipe installation is not detected you need to set the cmake variables *IPE_INCLUDE_DIR* and *IPE_LIBRARIES*. If you want to install ipelets automatically (using *make install* for example) you also need to set the cmake variable *IPELET_INSTALL_DIR*, if the ipelet installation path has not already been detected. These ipelets have been tested with the versions 6.0pre28 to 6.0pre32 of Ipe, as well as with the version 7 (at least up to version 7.10). These ipelets are a good starting point to design your own ipelet. For more details on how to compile an ipelet, please refer to the Ipe manual.

82.5 Design and Implementation History

A first draft of the CGAL ipelets has been implemented during the internship of Nicolas Carrez, in 2005, supervised by Olivier Devillers. Since then, the ipelets have been rewritten and extended by Sebastien Loriot as a stand-alone software with the help of Laurent Rineau. Finally, an enhanced version has been formally contributed to CGAL in 2009 by Sebastien Loriot. Sylvain Pion has participated in the discussion of the overall design, and contributed the Hilbert sort ipelet demo.

CGAL Ipelets Reference Manual

Sébastien Loriot and Sylvain Pion

Classes

82.6 Alphabetical List of Reference Pages

CGAL::Ipelet_base<Kernel,int nbf>

Definition

Ipelet_base<Kernel,int nbf> is an abstract base class for defining an ipelet. The only function that needs to be defined in a derived class is *protected_run(int i)* that contains the code of an ipelet. Note that the name of the function suggests that the ipelet may throw exceptions that will be caught by a function of the class *Ipelet_base<Kernel,int nbf>* avoiding Ipe to crash.

Parameters

The *Kernel* template parameter determines the kernel that will be used in the ipelet. This parameter must be set according to the algorithm to be used. The integer *nbf* indicates the number of functions defined by the ipelet.

Types

typedef Kernel	*Kernel;*	The kernel used to define internal types.
typedef Kernel::FT	*FT;*	The number type of the coordinates.
typedef Kernel::Point_2		
	Point_2;	The point type.
typedef Kernel::Circle_2		
	Circle_2;	The circle type.
typedef CGAL::tuple<Circle_2,Point_2,Point_2,CGAL::Sign>		
	Circular_arc_2;	The circular arc type. The *CGAL::Sign*, equals either to *CGAL::COUNTERCLOCKWISE* or *CGAL::CLOCKWISE*, indicates if the arc is the set of points on the circle from the first point to the second point turning clockwise or counterclockwise.
typedef CGAL::Weighted_point<Point,FT>		
	Weighted_point_2;	The weighted point type.
typedef Kernel::Segment_2		
	Segment_2;	The segment type.
typedef Kernel::Line_2		
	Line_2;	The line type.

typedef Kernel::Ray_2

Ray_2; The ray type.

typedef Kernel::Triangle_2

Triangle_2; The triangle type.

typedef CGAL::Polygon_2<Kernel>

Polygon_2; The polygon type.

typedef Kernel::Iso_rectangle_2

Iso_rectangle_2; A type to represent bounding boxes.

The set of *read objects* is defined as the set of objects of type *Point_2*, *Segment_2*, *Polygon_2*, *Circular_arc_2* or *Circle_2*. The set of *drawable objects* is defined as the super set of the set of *read objects* also including objects of type *Line_2*, *Ray_2*, *Triangle_2* and *Iso_rectangle_2*.

template<class output_iterator>

Point_grabber; Class type providing operators to extract points from segments and polygons.

template<class output_iterator>

Segment_grabber; Class type providing operators to extract segments from polygons.

Creation

Ipelet_base<Kernel,int nbf> iplt(const std::string fct_names[], const std::string help_msg[], const std::string name)

initializes an ipelet. The string *name* is the name given to the ipelet in the Ipe menu. The string array *fct_names* contains the names of these functions as they will appear in the submenu of the ipelet. The string array *help_msg* of size one or *nbf-1* contains a help message for each function of the ipelet. These help messages can be printed using the member function *show_help()*. This function expects that the last function defined in the ipelet is dedicated to print the help message. It is advised that the last function defined is dedicated to show a help message.

Access Functions

advanced

*IpePage** *iplt.get_ipe_page()* returns a pointer to an Ipe object representing the drawing page. Refer to the Ipe library documentation for more details.

*IpeletHelper** *iplt.get_ipelet_helper()*

returns a pointer to an Ipe object providing services to Ipelets. Refer to the Ipe library documentation for more details.

advanced

Operations

void *iplt.show_help(bool one_per_func=true)*

Prints in Ipe a pop-up help message constructed from the string array of the constructor. This function expects that the last function defined in the ipelet is dedicated to print the help message. When the boolean *one_per_func* is *true*, one help message per function is printed (except for the help function itself) using the *nbf-1* strings in the array given to the constructor. Otherwise, only one help message is printed using the first string in the array given to the constructor.

void *iplt.print_error_message(std::string msg)*

Prints the string *msg* as a message in Ipe.

void *iplt.protected_run(int i) =0*

function called when the user selects the CGAL ipelet.

template< class V, class O >
Iso_rectangle_2 *iplt.read_active_objects(Dispatch_or_drop_output_iterator<V, O> out, bool deselect_all=true, bool delete_selected_objects=false)*

This function assigns to output iterator *out* all handled objects selected in Ipe. Objects read belong to the set of *read objects* (even within groups). The output iterator *out* must be able to handle all *read objects* (see *CGAL::Dispatch_or_drop_output_iterator<V,O>*). In addition, a bounding box (*Iso_rectangle_2*) of the active objects selected is returned. The two Boolean arguments indicate whether the retrieved objects must be deselected and/or removed. Note that non-retrieved objects (primitives of Ipe not handled or objects dropped by *out*) are automatically deselected, and not deleted. If a non-retrieved object is a sub-path or is inside a group, the whole path or group will not be deleted.

template<class Output_iterator>

boost::function_output_iterator<Point_grabber<Output_iterator> >

iplt.point_grabber(Output_iterator it)

returns an output iterator which wraps *it*. *Output_iterator* must be a model of the output iterator concept accepting assignments from *Point_2*. The returned output iterator will accept assignments from objects of types *Polygon_2* or *Segment_2* or *Point_2*, it decomposes them in objects of type *Point_2* and assigns them to *it*. For more details on the returned output iterator refer to the Boost library documentation.

template<class Output_iterator>
boost::function_output_iterator<Segment_grabber<Output_iterator> >

iplt.segment_grabber(Output_iterator it)

returns an output iterator which wraps *it*. *Output_iterator* must be a model of the output iterator concept accepting assignments from *Segment_2*. The returned output iterator will accept assignments from objects of types *Polygon_2* or *Segment_2*, it decomposes them in objects of type *Segment_2* and assigns them to *it*. For more details on the returned output iterator refer to the Boost library documentation.

template<class T>
void *iplt.draw_in_ipe(T object, bool deselect=false)*

This function draws in the page of Ipe a given object. *T* must be a type inside the set of *drawable objects*. When *object* is of type *Line_2* or *Ray_2*, only the part of the object that is inside the page of Ipe (if not empty) is drawn. This function is also able to draw a 2D CGAL triangulation as a group of segments. If the boolean *deselect* is set to *true*, object drawn is deselected.

template<class T>
void *iplt.draw_in_ipe(T object, Iso_rectangle_2 bbox, bool deselect=false)*

Same as above, except that objects are clipped to *bbox* before been drawn.

template<class Iterator>
void *iplt.draw_in_ipe(Iterator begin,*
Iterator end,
bool makegrp=true,
bool deselectall=false)

This function draws in the page of Ipe a set of objects given by an iterator range. These objects must be of a type inside the set of *drawable objects*. If the boolean *makegrp* is set to *true*, objects drawn define a group in Ipe. If the boolean *deselectall* is set to *true*, objects drawn are deselected.

template<class Iterator>
void *iplt.draw_in_ipe(Iterator begin, Iterator end, Iso_rectangle_2 bbox, bool makegrp=true, bool deselectall=false)*

Same as above, except that objects are clipped to *bbox* before been drawn.

template<class iterator>
void *iplt.draw_polyline_in_ipe(iterator first, iterator last, bool setclose=false, bool deselect=false)*

This function draws in the page of Ipe a polyline defined by an iterator range of points. If the boolean *setclose* is *true*, the polyline drawn is closed. If the boolean *deselect* is set to *true*, polyline drawn is deselected.

template<class Triangulation>
void *iplt.draw_dual_in_ipe(Triangulation T, Iso_rectangle_2 bbox, bool makegrp=true, bool deselect=false)*

This function draws in the page of Ipe the dual of a 2D CGAL triangulation. The edges of the dual are restricted to the interior of *bbox*. If the boolean *makegrp* is set to *true*, segments drawn define a group in Ipe. If the boolean *deselect* is set to *true*, segments drawn are deselected.

template<class Triangulation>
void *iplt.draw_skeleton_in_ipe(Triangulation T, Iso_rectangle_2 bbox, bool makegrp=true, bool deselect=false)*

This function draws in the page of Ipe the Voronoi segment skeleton from a triangulation of segments. The edges are restricted to the interior of *bbox*. If the boolean *makegrp* is set to *true*, segments drawn define a group in Ipe. If the boolean *deselect* is set to *true*, segments drawn are deselected.

template <class T>

std::pair<int,T> *iplt.request_value_from_user(std::string msg)*

This function induces the creation of a dialog box requesting a value from the user. The string *msg* is printed in this dialog box. Ipe lexer tries to convert the user input into an object of type *T* (a simple type such as *int*, *float*, etc). If the conversion is possible, the value is returned within a *std::pair*. The integer of the pair returned is -1 if the user input is not correct, 0 if user input is empty and 1, otherwise. otherwise.

CGAL_IPELET

The registration of a new ipelet can be done using the macro command *CGAL_IPELET*. Taking as a parameter the name of the class defining the new ipelet, that macro must be placed in the source file after the class definition.

Bibliography

[AA95] O. Aichholzer and F. Aurenhammer. Straight skeletons for general polygonal figures. Technical Report 432, Inst. for Theor. Comput. Sci., Graz Univ. of Technology, Graz, Austria, 1995.

[AAAG95] O. Aichholzer, D. Alberts, F. Aurenhammer, and B. Gärtner. A novel type of skeleton for polygons. *J. Universal Comput. Sci.*, 1(12):752–761, 1995.

[AAD07] Nina Amenta, Dominique Attali, and Olivier Devillers. Complexity of Delaunay triangulation for points on lower-dimensional polyhedra. In *Proc. 18th ACM-SIAM Sympos. Discrete Algorithms*, pages 1106–1113, 2007.

[AB99] Nina Amenta and Marshall Bern. Surface reconstruction by Voronoi filtering. *Discrete Comput. Geom.*, 22(4):481–504, 1999.

[AB03] Dominique Attali and Jean-Daniel Boissonnat. Complexity of the Delaunay triangulation of points on polyhedral surfaces. *Discrete and Computational Geometry*, 30(3):437–452, 2003.

[Abb] J. Abbott. Quadratic interval refinement for real roots. Poster presented at the 2006 Int. Symp. on Symb. and Alg. Comp. (ISSAC 2006).

[ABL03] Dominique Attali, Jean-Daniel Boissonnat, and André Lieutier. Complexity of the Delaunay triangulation of points on surfaces: The smooth case. In *Proc. 19th Annual Symposium on Computational Geometry*, pages 237–246, 2003.

[ACR03] Nina Amenta, Sunghee Choi, and Günter Rote. Incremental constructions con BRIO. In *Proc. 19th Annu. Sympos. Comput. Geom.*, pages 211–219, 2003.

[ACSYD05] Pierre Alliez, David Cohen-Steiner, Mariette Yvinec, and Mathieu Desbrun. Variational tetrahedral meshing. *ACM Transactions on Graphics*, 24:617–625, 2005. SIGGRAPH '2005 Conference Proceedings.

[AFH02] P. K. Agarwal, E. Flato, and D. Halperin. Polygon decomposition for efficient construction of Minkowski sums. *Computational Geometry: Theory and Applications*, 21:39–61, 2002.

[AKM$^+$87] A. Aggarwal, M. M. Klawe, S. Moran, P. W. Shor, and R. Wilber. Geometric applications of a matrix-searching algorithm. *Algorithmica*, 2:195–208, 1987.

[Ale01] Andrei Alexandrescu. *Modern C++ Design: Generic Programming and Design Patterns Applied*. Addison-Wesley, 2001.

[AM93a] S. Arya and D. M. Mount. Algorithms for fast vector quantization. In *Data Compression Conference*, pages 381–390. IEEE Press, 1993.

[AM93b] S. Arya and D. M. Mount. Approximate nearest neighbor queries in fixed dimensions. In *Proc. 4th ACM-SIAM Sympos. Discrete Algorithms*, pages 271–280, 1993.

[And78] K. R. Anderson. A reevaluation of an efficient algorithm for determining the convex hull of a finite planar set. *Inform. Process. Lett.*, 7(1):53–55, 1978.

[And79] A. M. Andrew. Another efficient algorithm for convex hulls in two dimensions. *Inform. Process. Lett.*, 9(5):216–219, 1979.

[AS00] Pankaj K. Agarwal and Micha Sharir. Arrangements and their applications. In Jörg-Rüdiger Sack and Jorge Urrutia, editors, *Handbook of Computational Geometry*, pages 49–119. Elsevier Science Publishers B.V. North-Holland, Amsterdam, 2000.

[AT78] S. G. Akl and G. T. Toussaint. A fast convex hull algorithm. *Inform. Process. Lett.*, 7(5):219–222, 1978.

[Aus98] Matthew H. Austern. *Generic Programming and the STL*. Addison-Wesley, 1998.

[Bau75] B. G. Baumgart. A polyhedron representation for computer vision. In *Proc. AFIPS Natl. Comput. Conf.*, volume 44, pages 589–596. AFIPS Press, Alrington, Va., 1975.

[BB97] F. Bernardini and C. Bajaj. Sampling and reconstructing manifolds using alpha-shapes. Technical Report CSD-TR-97-013, Dept. Comput. Sci., Purdue Univ., West Lafayette, IN, 1997.

[BBP01] H. Brönnimann, C. Burnikel, and S. Pion. Interval arithmetic yields efficient dynamic filters for computational geometry. *Discrete Applied Mathematics*, 109:25–47, 2001.

[BDH96] C. Bradford Barber, David P. Dobkin, and Hannu Huhdanpaa. The Quickhull algorithm for convex hulls. *ACM Trans. Math. Softw.*, 22(4):469–483, December 1996.

[BDP$^+$02] Jean-Daniel Boissonnat, Olivier Devillers, Sylvain Pion, Monique Teillaud, and Mariette Yvinec. Triangulations in CGAL. *Comput. Geom. Theory Appl.*, 22:5–19, 2002.

[BDTY00] Jean-Daniel Boissonnat, Olivier Devillers, Monique Teillaud, and Mariette Yvinec. Triangulations in CGAL. In *Proc. 16th Annu. ACM Sympos. Comput. Geom.*, pages 11–18, 2000.

[BEH$^+$02] Eric Berberich, Arno Eigenwillig, Michael Hemmer, Susan Hert, Kurt Mehlhorn, and Elmar Schömer. A computational basis for conic arcs and boolean operations on conic polygons. In Rolf Möhring and Rajeev Raman, editors, *Algorithms - ESA 2002: 10th Annual European Symposium*, volume 2461 of *Lecture Notes in Computer Science*, pages 174–186, Rome, Italy, September 2002. Springer.

[BEH$^+$05] Eric Berberich, Arno Eigenwillig, Michael Hemmer, Susan Hert, Lutz Kettner, Kurt Mehlhorn, Joachim Reichel, Susanne Schmitt, Elmar Schömer, and Nicola Wolpert. Exacus: Efficient and exact algorithms for curves and surfaces. In Gerth S. Brodal and Stefano Leonardi, editors, *13th Annual European Symposium on Algorithms (ESA 2005)*, volume 3669 of *Lecture Notes in Computer Science*, pages 155–166, Palma de Mallorca, Spain, October 2005. European Association for Theoretical Computer Science (EATCS), Springer.

[Ben75] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, September 1975.

[BEPP99] Hervé Brönnimann, Ioannis Emiris, Victor Pan, and Sylvain Pion. Sign determination in Residue Number Systems. *Theoret. Comput. Sci.*, 210(1):173–197, 1999. Special Issue on Real Numbers and Computers.

[BF02] Jean-Daniel Boissonnat and Julia Flötotto. A local coordinate system on a surface. In *Proc. 7th ACM Symposium on Solid Modeling and Applications*, 2002.

[BFH95] Heinzgerd Bendels, Dieter W. Fellner, and Sven Havemann. Modellierung der grundlagen: Erweiterbare datenstrukturen zur modellierung und visualisierung polygonaler welten. In D. W. Fellner, editor, *Modeling – Virtual Worlds – Distributed Graphics*, pages 149–157, Bad Honnef / Bonn, 27.–28. November 1995.

[BG89] D. Bertsimas and M. Grigni. On the space-filling curve heuristic for the euclidean traveling salesman problem. *Operations Research Letters*, 8:241–244, 1989.

[BGH97] Julien Basch, Leonidas Guibas, and John Hershberger. Data structures for mobile data. In *Proceedings of the 8th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 747–756, 1997.

[BHKT07] Eric Berberich, Michael Hemmer, Menelaos I. Karavelas, and Monique Teillaud. Revision of the interface specification of algebraic kernel. Technical Report ACS-TR-243301-01, INRIA Sophia-Antipolis, Max Planck Institut für Informatik, National University of Athens, 2007.

[BMS94] C. Burnikel, K. Mehlhorn, and S. Schirra. On degeneracy in geometric computations. In *Proc. 5th ACM-SIAM Sympos. Discrete Algorithms*, pages 16–23, 1994.

[BO05] Jean-Daniel Boissonnat and Steve Oudot. Provably good sampling and meshing of surfaces. *Graphical Models*, 67:405–451, 2005.

[BPP95] Gavin Bell, Anthony Parisi, and Mark Pesce. Vrml the virtual reality modeling language: Version 1.0 specification. `http://www.vrml.org/`, May 26 1995. Third Draft.

[Bro97] J. L. Brown. Systems of coordinates associated with points scattered in the plane. *Comput. Aided Design*, 14:547–559, 1997.

[Bur96] C. Burnikel. *Exact Computation of Voronoi Diagrams and Line Segment Intersections*. Ph.D thesis, Universität des Saarlandes, March 1996.

[But71] Arthur Butz. Alternative algorithm for Hilbert's space-filling curve. *IEEE Transactions on computers*, pages 424–425, 1971.

[BY98] Jean-Daniel Boissonnat and Mariette Yvinec. *Algorithmic Geometry*. Cambridge University Press, UK, 1998. Translated by Hervé Brönnimann.

[BYB09a] Dobrina Boltcheva, Mariette Yvinec, and Jean-Daniel Boissonnat. Feature preserving delaunay mesh generation from 3d multi- material images. *Computer Graphics Forum*, 28:1455–14645, 2009. special issue for EUROGRAPHICS Symposium on Geometry Processing.

[BYB09b] Dobrina Boltcheva, Mariette Yvinec, and Jean-Daniel Boissonnat. Mesh generation from 3d multi-material images. In *Medical Image Computing and Computer-Assisted Intervention*, volume 5762 of *Lecture Notes in Computer Science*, pages 283–290, 2009.

[Byk78] A. Bykat. Convex hull of a finite set of points in two dimensions. *Inform. Process. Lett.*, 7:296–298, 1978.

[C++98] International standard ISO/IEC 14882: Programming languages – C++. American National Standards Institute, 11 West 42nd Street, New York 10036, 1998.

[CC78] E. Catmull and J. Clark. Recursively generated B-spline surfaces on arbitrary topological meshes. *Computer Aided Design*, 10:350–355, 1978.

[CD85] Bernard Chazelle and D. P. Dobkin. Optimal convex decompositions. In G. T. Toussaint, editor, *Computational Geometry*, pages 63–133. North-Holland, Amsterdam, Netherlands, 1985.

[CDE+00] Siu-Wing Cheng, Tamal K. Dey, Herbert Edelsbrunner, Michael A. Facello, and Shang-Hua Teng. Sliver exudation. *J. ACM*, 47(5):883–904, 2000.

[CDL07] S.-W. Cheng, T. K. Dey, and J. A. Levine. A practical delaunay meshing algorithm for a large class of domains. In *Meshing Roundtable*, pages 477–494, 2007.

[CDR07] Siu-Wing Cheng, Tamal K. Dey, and Edgar A. Ramos. Delaunay refinement for piecewise smooth complexes. In *SODA*, pages 1096–1105, Philadelphia, PA, USA, 2007.

[Cha84] Bernard Chazelle. Convex partitions of polyhedra: a lower bound and worst-case optimal algorithm. *SIAM J. Comput.*, 13:488–507, 1984.

[Che93] L. P. Chew. Guaranteed-quality mesh generation for curved surfaces. In *Proc. 9th Annu. ACM Sympos. Comput. Geom.*, pages 274–280, 1993.

[Che04] L. Chen. Mesh Smoothing Schemes based on Optimal Delaunay Triangulations. In *Proceedings of 13th International Meshing Roundtable*, pages 109–120, 2004.

[Che09] Otfried Cheong. *IPE manual and library documentation*, 6.0pre32 edition, 2009.

[CLRS01] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, Cambridge, MA, 2nd edition, 2001.

[CMS93] K. L. Clarkson, K. Mehlhorn, and R. Seidel. Four results on randomized incremental constructions. *Comput. Geom. Theory Appl.*, 3(4):185–212, 1993.

[CP05a] F. Cazals and M. Pouget. Estimating differential quantities using polynomial fitting of osculating jets. *Computer Aided Geometric Design*, 22(2), 2005. Conference version: Symp. on Geometry Processing 2003.

[CP05b] F. Cazals and M. Pouget. Smooth surfaces, umbilics, lines of curvatures, foliations, ridges and the medial axis: selected topics. *Int. J. of Computational Geometry and Applications*, 15(5):511–536, 2005.

[CP05c] F. Cazals and M. Pouget. Topology driven algorithms for ridge extraction on meshes. Technical Report RR-5526, INRIA, 2005.

[CT09] Manuel Caroli and Monique Teillaud. Computing 3D periodic triangulations. In *Proceedings 17th European Symposium on Algorithms*, volume 5757 of *Lecture Notes in Computer Science*, pages 37–48, 2009. Full version available as INRIA Research Report 6823 `http://hal.inria.fr/inria-00356871`.

[Dam10] G. Damiand. *Contributions aux Cartes Combinatoires et Cartes Généralisées : Simplification, Modèles, Invariants Topologiques et Applications*. Habilitation à diriger des recherches, Université Lyon 1, Septembre 2010.

[dBvKOS97] Mark de Berg, Marc van Kreveld, Mark Overmars, and Otfried Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, Berlin, 1997.

[dBvKOS00] Mark de Berg, Marc van Kreveld, Mark Overmars, and Otfried Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, Berlin, Germany, 2nd edition, 2000.

[dC76] M. de Carmo. *Differential Geometry of Curves and Surfaces*. Prentice Hall, Englewood Cliffs, NJ, 1976.

[dCCLT09] Pedro M.M. de Castro, Frederic Cazals, Sebastien Loriot, and Monique Teillaud. Design of the CGAL 3D Spherical Kernel and application to arrangements of circles on a sphere. *Computational Geometry : Theory and Applications*, 42(6-7):536 – 550, 2009.

[DEGN99] Tamal Dey, Herbert Edelsbrunner, Sumanta Guha, and Dmitry Nekhayev. Topology preserving edge contraction. geometric combinatorics. *Publ. Inst. Math. (Beograd) (N.S.)*, 66:23–45, 1999.

[Dev98] Olivier Devillers. Improved incremental randomized Delaunay triangulation. In *Proc. 14th Annu. ACM Sympos. Comput. Geom.*, pages 106–115, 1998.

[Dev02] Olivier Devillers. The Delaunay hierarchy. *Internat. J. Found. Comput. Sci.*, 13:163–180, 2002.

[Dev09] Olivier Devillers. Vertex removal in two dimensional Delaunay triangulation: Asymptotic complexity is pointless. Research Report 7104, INRIA, 2009.

[DFG99] Q. Du, V. Faber, and M. Gunzburger. Centroidal Voronoi Tessellations: Applications and Algorithms. *SIAM review*, 41(4):637–676, 1999.

[DFMT00] Olivier Devillers, Alexandra Fronville, Bernard Mourrain, and Monique Teillaud. Algebraic methods and arithmetic filtering for exact predicates on circle arcs. In *Proc. 16th Annu. ACM Sympos. Comput. Geom.*, pages 139–147, 2000.

[DFMT02] Olivier Devillers, Alexandra Fronville, Bernard Mourrain, and Monique Teillaud. Algebraic methods and arithmetic filtering for exact predicates on circle arcs. *Comput. Geom. Theory Appl.*, 22:119–142, 2002.

[DLPT98] Olivier Devillers, Giuseppe Liotta, Franco P. Preparata, and Roberto Tamassia. Checking the convexity of polytopes and the planarity of subdivisions. *Comput. Geom. Theory Appl.*, 11:187–208, 1998.

[DMA02] Mathieu Desbrun, Mark Meyer, and Pierre Alliez. Intrinsic parameterizations of surface meshes. *Computer Graphics Forum*, 21(3):209–218, September 2002.

[DP03] Olivier Devillers and Sylvain Pion. Efficient exact geometric predicates for Delaunay triangulations. In *Proc. 5th Workshop Algorithm Eng. Exper.*, pages 37–44, 2003.

[DPT02] Olivier Devillers, Sylvain Pion, and Monique Teillaud. Walking in a triangulation. *Internat. J. Found. Comput. Sci.*, 13:181–199, 2002.

[DT03] Olivier Devillers and Monique Teillaud. Perturbations and vertex removal in a 3D Delaunay triangulation. In *Proc. 14th ACM-SIAM Sympos. Discrete Algorithms (SODA)*, pages 313–319, 2003.

[DT06] Olivier Devillers and Monique Teillaud. Perturbations and vertex removal in Delaunay and regular 3D triangulations. Research Report 5968, INRIA, 2006. `http://hal.inria.fr/inria-00090522`.

[DW02] Q. Du and D. Wang. Tetrahedral mesh generation and optimization based on centroidal Voronoi tessellations. *International Journal for Numerical Methods in Engineering*, 56:1355–1373, 2002.

[Dwy89] R. A. Dwyer. Higher-dimensional Voronoi diagrams in linear expected time. In *Proc. 5th Annu. ACM Sympos. Comput. Geom.*, pages 326–333, 1989.

[Edd77] W. F. Eddy. A new convex hull algorithm for planar sets. *ACM Trans. Math. Softw.*, 3:398–403 and 411–412, 1977.

[EDD+95] Matthias Eck, Tony DeRose, Tom Duchamp, Hugues Hoppe, Michael Lounsbery, and Werner Stuetzle. Multiresolution analysis of arbitrary meshes. In *Computer Graphics (Proc. SIGGRAPH '95)*, volume 29, pages 173–182, 1995. Examples in `ftp://ftp.cs.washington.edu/pub/graphics`.

[Ede92] H. Edelsbrunner. Weighted alpha shapes. Technical Report UIUCDCS-R-92-1760, Dept. Comput. Sci., Univ. Illinois, Urbana, IL, 1992.

[Ede99] H. Edelsbrunner. Deformable smooth surface design. *Discrete Comput. Geom.*, 21:87–115, 1999.

[EE98] David Eppstein and Jeff Erickson. Raising roofs, crashing cycles, and playing pool: Applications of a data structure for finding pairwise interactions. In *Symposium on Computational Geometry*, pages 58–67, 1998.

[Eig08] Arno Eigenwillig. *Real Root Isolation for Exactand Approximate Polynomials Using Descartes ' Rule of Signs*. PhD thesis, Universität des Saarlandes, Saarbrücken, Germany, 2008.

[EK99] G. Elber and M.-S. Kim, editors. *Computer Aided Design*, volume 31, 1999. Special Issue on Offsets, Sweeps and Minkowski Sums.

[EK08] Arno Eigenwillig and Michael Kerber. Exact and efficient 2d-arrangements of arbitrary algebraic curves. In *Proceedings of the Nineteenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA08)*, 2008. 122–131.

[EKP+04] Ioannis Z. Emiris, Athanasios Kakargias, Sylvain Pion, Monique Teillaud, and Elias P. Tsigaridas. Towards an open curved kernel. In *Proc. 20th Annu. ACM Sympos. Comput. Geom.*, pages 438–446, 2004.

[EKW07] Arno Eigenwillig, Michael Kerber, and Nicola Wolpert. Fast and exact geometric analysis of real algebraic plane curves. In Christopher W. Brown, editor, *Proocedings of the 2007 International Symposium on Symbolic and Algebraic Computation (ISSAC 2007)*, pages 151–158, 2007.

[EM94] H. Edelsbrunner and E. P. Mücke. Three-dimensional alpha shapes. *ACM Trans. Graph.*, 13(1):43–72, January 1994.

[Eri02] Jeff Erickson. Dense point sets have sparse Delaunay triangulations. In *Proc. 13th ACM-SIAM Sympos. Discrete Algorithms (SODA)*, pages 125–134, 2002.

[ES96] H. Edelsbrunner and N. R. Shah. Incremental topological flipping works for regular triangulations. *Algorithmica*, 15:223–241, 1996.

[Far90] G. Farin. Surfaces over Dirichlet tesselations. *Comput. Aided Geom. Design*, 7:281–292, 1990.

[FBF77] J. H. Friedman, J. L. Bentley, and R. A. Finkel. An algorithm for finding best matches in logarithmic expected time. *ACM Trans. Math. Softw.*, 3:209–226, 1977.

[FH00] Eyal Flato and Dan Halperin. Robust and efficient construction of planar Minkowski sums. In *Abstracts 16th European Workshop Comput. Geom.*, pages 85–88. Ben-Gurion University of the Negev, 2000.

[FH05] M. S. Floater and K. Hormann. Surface parameterization: a tutorial and survey. In N. A. Dodgson, M. S. Floater, and M. A. Sabin, editors, *Advances in Multiresolution for Geometric Modelling*, Mathematics and Visualization, pages 157–186. Springer, Berlin, Heidelberg, 2005.

[FJ83] G. N. Frederickson and D. B. Johnson. Finding *k*th paths and *p*-centers by generating and searching good data structures. *J. Algorithms*, 4:61–80, 1983.

[FJ84] G. N. Frederickson and D. B. Johnson. Generalized selection and ranking: sorted matrices. *SIAM J. Comput.*, 13:14–30, 1984.

[Flo03a] Michael Floater. Mean value coordinates. *Computer Aided Design*, 20(1):19–27, 2003.

[Flö03b] Julia Flötotto. *A coordinate system associated to a point cloud issued from a manifold: definition, properties and applications*. Thèse de doctorat en sciences, Université de Nice-Sophia Antipolis, France, 2003.

[FO98] Petr Felkel and Stěpán Obdržálek. Straight skeleton implementation. In László Szirmay Kalos, editor, *14th Spring Conference on Computer Graphics (SCCG'98)*, pages 210–218, 1998.

[Gär99] B. Gärtner. Fast and robust smallest enclosing balls. In *Proc. 7th annu. European Symposium on Algorithms (ESA)*, volume 1643 of *Lecture Notes in Computer Science*, pages 325–338. Springer-Verlag, 1999.

[GGHT97] M. Goodrich, L. J. Guibas, J. Hershberger, and P. Tanenbaum. Snap rounding line segments efficiently in two and three dimensions. In *Proc. 13th Annu. ACM Sympos. Comput. Geom.*, pages 284–293, 1997.

[GH97] M. Garland and P. S. Heckbert. Surface simplification using quadric error metrics. In *Proc. SIGGRAPH '97*, pages 209–216, 1997.

[GHH+03] Miguel Granados, Peter Hachenberger, Susan Hert, Lutz Ketter, Kurt Mehlhorn, and Michael Seel. Boolean operations on 3d selective nef complexes data structure, algorithms, and implementation. In Giuseppe Di Battista and Uri Zwick, editors, *Algorithms - ESA 2003: 11th Annual European Symposium*, volume 2832 of *Lecture Notes in Computer Science*, pages 174–186, Budapest, Hugary, September 2003. Springer.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[GKR04] Leonidas Guibas, Menelaos Karaveles, and Daniel Russel. A computational framework for handling motion. In *Proceedings of the Sixth Workshop on Algorithm Engineering and Experiments*, pages 129–141, 2004.

[GM98] Leonidas Guibas and David Marimont. Rounding arrangements dynamically. *Internat. J. Comput. Geom. Appl.*, 8:157–176, 1998.

[Gra] Torbjörn Granlund. GMP, the GNU multiple precision arithmetic library. `http://gmplib.org/`.

[Gra72] R. L. Graham. An efficient algorithm for determining the convex hull of a finite planar set. *Inform. Process. Lett.*, 1:132–133, 1972.

[Gre83] Daniel H. Greene. The decomposition of polygons into convex parts. In Franco P. Preparata, editor, *Computational Geometry*, volume 1 of *Adv. Comput. Res.*, pages 235–259. JAI Press, Greenwich, Conn., 1983.

[GRS83] Leonidas J. Guibas, L. Ramshaw, and J. Stolfi. A kinetic framework for computational geometry. In *Proc. 24th Annu. IEEE Sympos. Found. Comput. Sci.*, pages 100–111, 1983.

[GS78] Leonidas J. Guibas and R. Sedgewick. A dichromatic framework for balanced trees. In *Proc. 19th Annu. IEEE Sympos. Found. Comput. Sci.*, Lecture Notes Comput. Sci., pages 8–21. Springer-Verlag, 1978.

[GS85] Leonidas J. Guibas and J. Stolfi. Primitives for the manipulation of general subdivisions and the computation of Voronoi diagrams. *ACM Trans. Graph.*, 4(2):74–123, April 1985.

[GS97a] B. Gärtner and S. Schönherr. Exact primitives for smallest enclosing ellipses. In *Proc. 13th Annu. ACM Sympos. Comput. Geom.*, pages 430–432, 1997.

[GS97b] Bernd Gärtner and Sven Schönherr. Smallest enclosing ellipses – fast and exact. Serie B – Informatik B 97-03, Freie Universität Berlin, Germany, June 1997. URL `http://www.inf.fu-berlin.de/inst/pubs/tr-b-97-03.abstract.html`.

[GS98a] Bernd Gärtner and Sven Schönherr. Smallest enclosing circles – an exact and generic implementation in C++. Serie B – Informatik B 98-04, Freie Universität Berlin, Germany, April 1998. URL `http://www.inf.fu-berlin.de/inst/pubs/tr-b-98-04.abstract.html`.

[GS98b] Bernd Gärtner and Sven Schönherr. Smallest enclosing ellipses – an exact and generic implementation in C++. Serie B – Informatik B 98-05, Freie Universität Berlin, Germany, April 1998. URL `http://www.inf.fu-berlin.de/inst/pubs/tr-b-98-05.abstract.html`.

[GS00] Bernd Gärtner and Svend Schönherr. An efficient, exact, and generic quadratic programming solver for geometric optimization. In *Proc. 16th Annu. ACM Sympos. Comput. Geom.*, pages 110–118, 2000.

[GvL83] G. Golub and C. van Loan. *Matrix Computations*. Johns Hopkins Univ. Press, Baltimore, MA, 1983.

[GVRLR98] L. Gonzalez-Vega, T. Recio, H. Lombardi, and M.-F. Roy. Sturm-habicht sequences, determinants and real roots of univariate polynomials. In B.F. Caviness and J.R. Johnson, editors, *Quantifier Elimination and Cylindrical Algebraic Decomposition*, Texts and Monographs in Symbolic Computation, pages 300–316. Springer, 1998.

[Hal04] Dan Halperin. Arrangements. In Jacob E. Goodman and Joseph O'Rourke, editors, *Handbook of Discrete and Computational Geometry*, chapter 24, pages 529–562. Chapman & Hall/CRC, 2nd edition, 2004.

[Han91] E. N. Hanson. The interval skip list: a data structure for finding all intervals that overlap a point. In *Proc. 2nd Workshop Algorithms Data Struct.*, volume 519 of *Lecture Notes Comput. Sci.*, pages 153–164. Springer-Verlag, 1991.

[HDD+92] Hugues Hoppe, Tony DeRose, Tom Duchamp, John McDonald, and Werner Stuetzle. Surface reconstruction from unorganized points. In *Computer Graphics (Proc. SIGGRAPH '90)*, volume 26, pages 71–77, 1992.

[HDD+93] H. Hoppe, T. DeRose, T. Duchamp, J. McDonald, and W. Stuetzle. Mesh optimization. In *Proc. SIGGRAPH '93*, pages 19–26, 1993.

[HGY+99] P. W. Hallinan, G. Gordon, A.L. Yuille, P. Giblin, and D. Mumford. *Two-and Three-Dimensional Patterns of the Face*. A.K.Peters, 1999.

[HH08] I. Haran and D. Halperin. An experimental study of point location in planar arrangements in cgal. *ACM Journal of Experimental Algorithmics*, 13, 2008.

[HKH12] M. Hemmer, M. Kleinbort, and D. Halperin. Improved Implementation of Point Location in General Two-Dimensional Subdivisions. *ArXiv e-prints*, May 2012.

[HM83] S. Hertel and K. Mehlhorn. Fast triangulation of simple polygons. In *Proc. 4th Internat. Conf. Found. Comput. Theory*, volume 158 of *Lecture Notes Comput. Sci.*, pages 207–218. Springer-Verlag, 1983.

[Hob99] J. D. Hobby. Practical segment intersection with finite precision output. *Comput. Geom. Theory Appl.*, 13(4):199–214, October 1999.

[Hof89a] C. Hoffmann. *Geometric and Solid Modeling*. Morgan-Kaufmann, San Mateo, CA, 1989.

[Hof89b] C. M. Hoffmann. The problems of accuracy and robustness in geometric computation. *IEEE Computer*, 22(3):31–41, March 1989.

[Hof89c] Christoph M. Hoffmann. *Geometric and Solid Modeling - An Introduction*. Morgan Kaufmann, 1989.

[Hof99] M. Hoffmann. A simple linear algorithm for computing rectangular three-centers. In *Proc. 11th Canad. Conf. Comput. Geom.*, pages 72–75, 1999.

[Hof04] Christoph M. Hoffmann. Solid modeling. In Jacob E. Goodman and Joseph O'Rourke, editors, *Handbook of Discrete and Computational Geometry*, chapter 56, pages 1257–1278. Chapman & Hall/CRC, 2nd edition, 2004.

[HP02] D. Halperin and E. Packer. Iterated snap rounding. *Computational Geometry: Theory and Applications*, 23(2):209–225, 2002.

[HS95] G. R. Hjaltason and H. Samet. Ranking in spatial databases. In M. J. Egenhofer and J. R. Herring, editors, *Advances in Spatial Databases - Fourth International Symposium*, number 951 in Lecture Notes Comput. Sci., pages 83–95, August 1995.

[HS00] Hisamoto Hiyoshi and Kokichi Sugihara. Voronoi-based interpolation with higher continuity. In *Proc. 16th Annu. ACM Sympos. Comput. Geom.*, pages 242–250, 2000.

[HW96] Jed Hartman and Josie Wernecke. *The VRML 2.0 Handbook: Building Moving Worlds on the Web*. Addison-Wesley, 1996.

[Jar73] R. A. Jarvis. On the identification of the convex hull of a finite set of points in the plane. *Inform. Process. Lett.*, 2:18–21, 1973.

[Jos99] Nicolai M. Josuttis. *The C++ Standard Library, A Tutorial and Reference*. Addison-Wesley, 1999.

[Kar04] Menelaos I. Karavelas. A robust and efficient implementation for the segment Voronoi diagram. In *Proc. Internat. Symp. on Voronoi diagrams in Science and Engineering (VD2004)*, pages 51–62, 2004.

[KBH06] Michael Kazhdan, M. Bolitho, and Hugues Hoppe. Poisson Surface Reconstruction. In *Symp. on Geometry Processing*, pages 61–70, 2006.

[KE02] Menelaos I. Karavelas and Ioannis Z. Emiris. Predicates for the planar additively weighted Voronoi diagram. Technical Report ECG-TR-122201-01, INRIA Sophia-Antipolis, Sophia-Antipolis, May 2002.

[KE03] Menelaos I. Karavelas and Ioannis Z. Emiris. Root comparison techniques applied to computing the additively weighted Voronoi diagram. In *Proc. 14th ACM-SIAM Sympos. Discrete Algorithms (SODA)*, pages 320–329, 2003.

[Ker09] Michael Kerber. *Geometric Algorithms for Algebraic Curves and Surfaces*. PhD thesis, Universität des Saarlandes, Saarbrücken, Germany, 2009.

[Ket98] L. Kettner. Designing a data structure for polyhedral surfaces. In *Proc. 14th Annu. ACM Sympos. Comput. Geom.*, pages 146–154, 1998.

[Ket99] L. Kettner. Using generic programming for designing a data structure for polyhedral surfaces. *Comput. Geom. Theory Appl.*, 13:65–90, 1999.

[Kha96] L. Khachiyan. Rounding of polytopes in the real number model of computation. *Mathematics of Operations Research*, 21(2):307–320, 1996.

[Kle89] Rolf Klein. *Concrete and Abstract Voronoi Diagrams*, volume 400 of *Lecture Notes Comput. Sci.* Springer-Verlag, 1989.

[KLPY99] Vijay Karamcheti, Chen Li, Igor Pechtchanski, and Chee Yap. *The CORE Library Project*, 1.2 edition, 1999. http://www.cs.nyu.edu/exact/core/.

[KM76] K. Kuratowski and A. Mostowski. *Set Theory*. North-Holland Publishing Co., 1976.

[KMP$^+$08] Lutz Kettner, Kurt Mehlhorn, Sylvain Pion, Stefan Schirra, and Chee Yap. Classroom examples of robustness problems in geometric computations. *Computational Geometry: Theory and Applications*, 40(1):61–78, may 2008.

[Kob00] Leif Kobbelt. $\sqrt{3}$-subdivision. In *Computer Graphics (Proc. SIGGRAPH '00)*, volume 34, pages 103–112, 2000.

[KV05] N.G.H. Kruithof and G. Vegter. Meshing skin surfaces with certified topology. In *Proceedings of the nineth International CADGraphics conference*, page to appear., 2005.

[KY02] Menelaos Karavelas and Mariette Yvinec. Dynamic additively weighted voronoi diagrams in 2d. In *Proc. 10th European Symposium on Algorithms*, pages 586–598, 2002.

[Lat91] J.-C. Latombe. *Robot Motion Planning*. Kluwer Academic Publishers, Boston, 1991.

[LD03] R. G. Laycock and A. M. Day. Automatically generating roof models from building footprints. In *The 11-th International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision'2003. Journal of WSCG - FULL Papers*, volume 11, 2003.

[Lev05] Bruno Levy. Numerical methods for digital geometry processing. In *Israel Korea Bi-National Conference - Invited talk, extended abstract (full paper will be available shortly)*, November 2005.

[Lie91] P. Lienhardt. Topological models for boundary representation: a comparison with n-dimensional generalized maps. *Computer-Aided Design*, 23(1):59–82, 1991.

[Lie94] P. Lienhardt. N-dimensional generalized combinatorial maps and cellular quasi-manifolds. *Internat. J. Comput. Geom. Appl.*, 4(3):275–324, 1994.

[LPRM02] Bruno Lévy, Sylvain Petitjean, Nicolas Ray, and Jérome Maillot. Least squares conformal maps for automatic texture atlas generation. In *Proceedings of the 29th Conference on Computer Graphics and Interactive Techniques SIGGRAPH*, volume 21(3) of *ACM Transactions on Graphics*, pages 362–371, 2002.

[LPT09] Sylvain Lazard, Luis Peñaranda, and Elias Tsigaridas. Univariate algebraic kernel and application to arrangements. In Jan Vahrenhold, editor, *SEA*, volume 5526 of *Lecture Notes in Computer Science*, pages 209–220. Springer, 2009.

[LS05] Yuanxin Liu and Jack Snoeyink. A comparison of five implementations of 3D delaunay tessellation. In János Pach Jacob E. Goodman and Emo Welzl, editors, *Combinatorial and Computational Geometry*, pages 439–458. MSRI Publications, 2005.

[LT98] Peter Lindstrom and Greg Turk. Fast and memory efficient polygonal simplification. In *IEEE Visualization*, pages 279–286, 1998.

[LT99] P. Lindstrom and G. Turk. Evaluation of memoryless simplification. *IEEE Transactions on Visualization and Computer Graphics*, 5(2):98–115, /1999.

[MAD05] Abdelkrim Mebarki, Pierre Alliez, and Olivier Devillers. Farthest point seeding for efficient placement of streamlines. In *Proceeding of IEEE Visualization*, 2005.

[Män88] M. Mäntylä. *An Introduction to Solid Modeling*. Computer Science Press, Rockville, MD, 1988.

[Meh84] Kurt Mehlhorn. *Data Structures and Algorithms 3: Multi-dimensional Searching and Computational Geometry*, volume 3 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, Heidelberg, Germany, 1984.

[Mel87] A. Melkman. On-line construction of the convex hull of a simple polyline. *Inform. Process. Lett.*, 25:11–12, 1987.

[Mey06] Michal Meyerovitch. Robust, generic, and efficient constructions of envelopes of surfaces in three-dimensional space. Master's thesis, Tel-Aviv University, Israel, 2006.

[MG06] J. Matoušek and B. Gärtner. *Understanding and Using Linear Programming*. Springer-Verlag, 2006.

[MN00] Kurt Mehlhorn and Stefan Näher. *LEDA: A Platform for Combinatorial and Geometric Computing*. Cambridge University Press, Cambridge, UK, 2000.

[MNS+96] Kurt Mehlhorn, Stefan Näher, Thomas Schilz, Stefan Schirra, Michael Seel, Raimund Seidel, and Christian Uhrig. Checking geometric programs or verification of geometric structures. In *Proc. 12th Annu. ACM Sympos. Comput. Geom.*, pages 159–165, 1996.

[MNSU] K. Mehlhorn, S. Näher, M. Seel, and C. Uhrig. *The LEDA User Manual*. Max-Planck-Insitut für Informatik, 66123 Saarbrücken, Germany. `http://www.mpi-sb.mpg.de/LEDA/leda.html`.

[MP78] D. E. Muller and F. P. Preparata. Finding the intersection of two convex polyhedra. *Theoret. Comput. Sci.*, 7:217–236, 1978.

[MP05] Guillaume Melquiond and Sylvain Pion. Formal certification of arithmetic filters for geometric predicates. In *Proc. 17th IMACS World Congress on Scientific, Applied Mathematics and Simulation*, 2005.

[MPFa] MPFI - the multiple precision floating-point interval library. Revol, Nathalie and Rouillier, Fabrice. `http://perso.ens-lyon.fr/nathalie.revol/software.html`.

[MPFb] MPFR - the multiple precision floating-point reliable library. The MPFR Team. `http://mpfr.org`.

[MS96] David R. Musser and Atul Saini. *STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library*. Addison-Wesley, 1996.

[MSW92] J. Matoušek, Micha Sharir, and Emo Welzl. A subexponential bound for linear programming. In *Proc. 8th Annu. ACM Sympos. Comput. Geom.*, pages 1–8, 1992.

[Mul90] K. Mulmuley. A fast planar partition algorithm, I. *J. Symbolic Comput.*, 10(3-4):253–280, 1990.

[Mye95] Nathan C. Myers. Traits: a new and useful template technique. *C++ Report*, June 1995.

[OBS04] Yutaka Ohtake, Alexander Belyaev, and Hans-Peter Seidel. Ridge-valley lines on meshes via implicit surface fitting. *ACM Trans. Graph.*, 23(3):609–612, 2004.

[Orl90] M. Orlowski. A new algorithm for the largest empty rectangle problem. *Algorithmica*, 5:65–73, 1990.

[ORY05] Steve Oudot, Laurent Rineau, and Mariette Yvinec. Meshing volumes bounded by smooth surfaces. In *Proc. 14th International Meshing Roundtable*, pages 203–219, 2005.

[PABL07] M. Poudret, A. Arnould, Y. Bertrand, and P. Lienhardt. Cartes combinatoires ouvertes. Research Notes 2007-1, Laboratoire SIC E.A. 4103, F-86962 Futuroscope Cedex, France, October 2007.

[PB89] L. K. Platzman and J. J. Bartholdi, III. Spacefilling curves and the planar travelling salesman problem. *J. ACM*, 36(4):719–737, October 1989.

[Ped70] D. Pedoe. *Geometry, a comprehensive course*. Dover Publications, New York, 1970.

[Pet01] S. Petitjean. A survey of methods for recovering quadrics in triangle meshes. *ACM Computing Surveys*, 34(2), 2001.

[Phi96] Mark Phillips. *Geomview Manual, Version 1.6.1 for Unix Workstations*. The Geometry Center, University of Minnesota, 1996. `http://www.geom.umn.edu/software/download/geomview.html`.

[Pip93] B. Piper. Properties of local coordinates based on dirichlet tesselations. *Computing Suppl.*, 8:227–239, 1993.

[Por01] I. Porteous. *Geometric Differentiation (2nd Edition)*. Cambridge University Press, 2001.

[PP93] U. Pinkall and K. Polthier. Computing discrete minimal surfaces and their conjugates. *Experimental Mathematics*, 2(1):15–36, 1993.

[PTVF02] W. Press, S. Teukolsky, W. Vetterling, and B. Flannery. *Numerical Recipes in C++*. Cambridge University Press, 2nd edition, 2002.

[Pug90] W. Pugh. Skip lists: a probabilistic alternative to balanced trees. *Commun. ACM*, 33(6):668–676, 1990.

[Req80] Aristides G. Requicha. Representations for rigid solids: Theory, methods, and systems. *ACM Computing Surveys*, 12(4):437–464, 1980.

[RS] RS - A software for real solving of algebraic systems. Rouillier, Fabrice. `http://www.loria.fr/equipes/vegas/rs/`.

[Rup95] J. Ruppert. A Delaunay refinement algorithm for quality 2-dimensional mesh generation. *J. Algorithms*, 18:548–585, 1995.

[RY07] Laurent Rineau and Mariette Yvinec. A generic software design for Delaunay refinement meshing. *Comput. Geom. Theory Appl.*, 38:100–110, 2007.

[RZ04] Fabrice Rouillier and Paul Zimmermann. Efficient isolation of polynomial's real roots. *Journal of Computational and Applied Mathematics*, 162(1):33–50, 2004.

[Sam90] H. Samet. *The Design and Analysis of Spatial Data Structures*. Addison-Wesley, Reading, MA, 1990.

[Sch95] O. Schwarzkopf. The extensible drawing editor Ipe. In *Proceedings of the eleventh annual symposium on Computational geometry*, pages 410–411. ACM New York, NY, USA, 1995.

[Sch96] Michael Schutte. Zufällige konvexe mengen. Master's thesis, Freie Universität Berlin, Germany, 1996.

[Sch00] Stefan Schirra. Robustness and precision issues in geometric computation. In Jörg-Rüdiger Sack and Jorge Urrutia, editors, *Handbook of Computational Geometry*, chapter 14, pages 597–632. Elsevier Science Publishers B.V. North-Holland, Amsterdam, 2000.

[Sei91] R. Seidel. A simple and fast incremental randomized algorithm for computing trapezoidal decompositions and for triangulating polygons. *Comput. Geom. Theory Appl.*, 1(1):51–64, 1991.

[She98a] Jonathan R. Shewchuk. A condition guaranteeing the existence of higher-dimensional constrained delaunay triangulations. In *Proc. 14th Annu. ACM Sympos. Comput. Geom.*, pages 76–85, 1998.

[She98b] Jonathan R. Shewchuk. Tetrahedral mesh generation by Delaunay refinement. In *Proc. 14th Annu. ACM Sympos. Comput. Geom.*, pages 86–95, 1998.

[She00] Jonathan R. Shewchuk. Mesh generation for domains with small angles. In *Proc. 16th Annu. ACM Sympos. Comput. Geom.*, pages 1–10, 2000.

[Sib80] R. Sibson. A vector identity for the Dirichlet tesselation. *Math. Proc. Camb. Phil. Soc.*, 87:151–155, 1980.

[Sib81] R. Sibson. A brief description of natural neighbour interpolation. In Vic Barnet, editor, *Interpreting Multivariate Data*, pages 21–36. John Wiley & Sons, Chichester, 1981.

[Sil97] Silicon Graphics Computer Systems, Inc. Standard template library programmer's guide. `http://www.sgi.com/Technology/STL/`, 1997.

[Skl72] J. Sklansky. Measuring concavity on rectangular mosaic. *IEEE Trans. Comput.*, C-21:1355–1364, 1972.

[SL95] Alexander Stepanov and Meng Lee. The standard template library, October 1995.

[SLL02] Jeremy G. Siek, Lie-Quan Lee, and Andrew Lumsdaine. *Boost Graph Library*. Addison-Wesley, 2002.

[SM00] M. Seel and K. Mehlhorn. Infimaximal frames: A technique for making lines look like segments. Research Report MPI-I-2000-1-005, MPI für Informatik, Saarbrücken, Germany, December 2000. `http://www.mpi-sb.mpg.de/~mehlhorn/ftp/InfiFrames.ps`.

[SP05] Le-Jeng Shiue and Jörg Peters. Mesh refinement based on euler encoding. In *Proceedings of the International Conference on Shape Modeling and Applications 2005*, pages 343–348, 2005.

[STV+95] Christian Schwarz, Jürgen Teich, Alek Vainshtein, Emo Welzl, and Brian L. Evans. Minimal enclosing parallelogram with application. In *Proc. 11th Annu. ACM Sympos. Comput. Geom.*, pages C34–C35, 1995.

[SW96] Micha Sharir and Emo Welzl. Rectilinear and polygonal p-piercing and p-center problems. In *Proc. 12th Annu. ACM Sympos. Comput. Geom.*, pages 122–132, 1996.

[Tar83] R. E. Tarjan. *Data Structures and Network Algorithms*, volume 44 of *CBMS-NSF Regional Conference Series in Applied Mathematics*. Society for Industrial and Applied Mathematics, Philadelphia, PA, 1983.

[Tei99] Monique Teillaud. Three dimensional triangulations in CGAL. In *Abstracts 15th European Workshop Comput. Geom.*, pages 175–178. INRIA Sophia-Antipolis, 1999.

[Ter05] P. Terdiman. OPCODE 3D Collision Detection library, 2005. http://www.codercorner.com/Opcode.htm.

[Tou83] G. T. Toussaint. Solving geometric problems with the rotating calipers. In *Proc. IEEE MELECON '83*, pages A10.02/1–4, 1983.

[Tou09] Jane Tournois. *Optimisation de maillages*. Thèse de doctorat en sciences, Université Nice Sophia-Antipolis, Nice, France, 2009.

[TSA09] Jane TOURNOIS, Rahul SRINIVASAN, and Pierre ALLIEZ. Perturbing slivers in 3D Delaunay meshes. In *Proceedings of the 18th International Meshing Roundtable*, october 2009.

[Tut63] W. T. Tutte. How to draw a graph. *Proceedings London Mathematical Society*, 13(52):743–768, 1963.

[TWAD09] Jane Tournois, Camille Wormser, Pierre Alliez, and Mathieu Desbrun. Interleaving Delaunay refinement and optimization for practical isotropic tetrahedron mesh generation. *ACM Transactions on Graphics*, 28(3):75:1–75:9, 2009. SIGGRAPH '2009 Conference Proceedings.

[Vai90] A. Vainshtein. Finding minimal enclosing parallelograms. *Diskretnaya Matematika*, 2:72–81, 1990. In Russian.

[vLS82] J. van Leeuwen and Anneke A. Schoone. Untangling a travelling salesman tour in the plane. In J. R. Mühlbacher, editor, *Proc. 7th Internat. Workshop Graph-Theoret. Concepts Comput. Sci.*, pages 87–98, München, 1982. Hanser.

[VRM96] The virtual reality modeling language specification: Version 2.0, ISO/IEC CD 14772. `http://www.vrml.org/`, August 4 1996.

[vzGG99] Joachim von zur Gathen and Jürgen Gerhard. *Modern Computer Algebra*. Cambridge University Press, 1999.

[Wei85] K. Weiler. Edge-based data structures for solid modeling in a curved surface environment. *IEEE Comput. Graph. Appl.*, 5(1):21–40, 1985.

[Wel91] Emo Welzl. Smallest enclosing disks (balls and ellipsoids). In H. Maurer, editor, *New Results and New Trends in Computer Science*, volume 555 of *Lecture Notes Comput. Sci.*, pages 359–370. Springer-Verlag, 1991.

[Wer94] Josie Wernicke. *The Inventor Mentor: Programming Object-Oriented 3D Graphics with Open Inventor, Release 2*. Addison-Wesley, 1994.

[WW02] Joe Warren and Henrik Weimer. *Subdivision Methods for Geometric Design*. Morgan Kaufmann Publishers, New York, 2002.

[Yap97] C. Yap. Towards exact geometric computation. *Comput. Geom. Theory Appl.*, 7(1):3–23, 1997.

[YD95] C. K. Yap and T. Dubé. The exact computation paradigm. In D.-Z. Du and F. K. Hwang, editors, *Computing in Euclidean Geometry*, volume 4 of *Lecture Notes Series on Computing*, pages 452–492. World Scientific, Singapore, 2nd edition, 1995.

[ZE02] Afra Zomorodian and Herbert Edelsbrunner. Fast software for box intersection. *Int. J. Comput. Geom. Appl.*, 12:143–172, 2002.

Index

Pages on which definitions are given are presented in **boldface**.